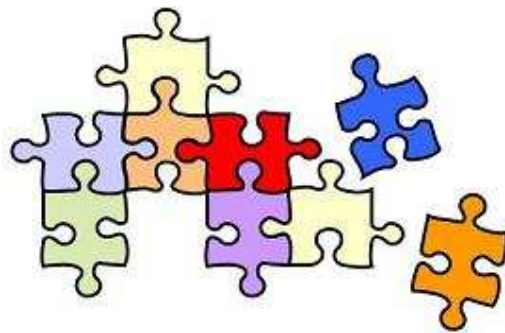


The Worklet Custom Service for YAWL

Installation and User Manual

Beta – 8 Release



Document Control

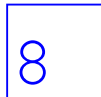
Date	Author	Version	Change
18 Dec 2005	Michael Adams	0.1	Initial Draft
3 March 2006	Lachlan Aldred	0.1.1	Merged installation manual
9 October 2006	Michael Adams	1.0	Updated for Beta 8 and the Exception Service

Preface

This manual contains instructions for installing & using the *Worklet Dynamic Process Selection & Exception Handling Custom Service for YAWL*.

Each section describes one part in the process of setting up and using the Worklet Service. It is probably best to work through the manual from start to finish the first time it is read. This manual focuses on the practical use of the Worklet Service. For those interested, a more technical description of the inner operations of worklets and the rule sets that support them can be found [here](#) or a more concise version [here](#).

All of the example specifications, rule sets, and so on referred to in this manual can be found in the “worklet repository” distributed with the service as part of the YAWL Beta 8 release.



This icon indicates a hands-on method or instruction.

Contents

Document Control	ii
Preface	ii
Contents	iii
Figures	iii
1. Welcome to the Worklet Service for YAWL.....	1
What is a Custom YAWL Service?	1
What is the YAWL Worklet Service?	1
The Selection Service	1
The Exception Service	2
2. Installation.....	3
Worklet Installation Package	3
Installing the Worklet Service.....	3
1. Extracting and Installing the Repository.....	3
2. Configuring the Service	4
3. Enabling the Worklet Exception Service	5
4. Installing the Rules Editor.....	6
3. Using the Worklet Selection Service	8
Top-level or Parent Specifications	8
Worklet Specifications.....	10
4. Using the Worklet Exception Service.....	13
Exception Types	14
Constraint Types.....	14
Externally Triggered Types.....	14
TimeOut	15
ItemAbort	15
ResourceUnavailable	15
ConstraintViolation.....	15
Exception Handling Primitives	15
5. Worklet Rule Sets and the Rules Editor.....	18
Worklet Rule Sets.....	18
The Rules Editor.....	21
Browsing an Existing Rule Set.....	22
The Toolbar	23
Other Features	24
Adding a New Rule	25
Adding a Conclusion – Selection Rule Type	28
Dynamic Replacement of an Executing Worklet.....	29
Creating a New Rule Set and/or Tree Set.....	31
Drawing a Conclusion Sequence.....	35
6. Walkthrough – Using the Worklet Service	37

A. Selection: Worklet-Enabled Atomic Task Example	37
B. Selection: Worklet-Enabled Multiple Instance Atomic Task Example	42
C. Exception: Constraints Example.....	46
D. Exception: External Trigger Example	49
E. Exception: Timeout Example	51
F: Rejecting a Worklet and/or Raising a New External Exception.....	54
Appendix A: Defining New Functions for Rule Node Conditions	57
Appendix B: Sample Log (generated by Walkthrough C).....	61

Figures

Figure 1: Worklet Repository Folder Structure	3
Figure 2: Contents of the workletService.war file	4
Figure 3: The Worklet Service's web.xml file (detail).....	5
Figure 4: The YAWL Engine's web.xml file (detail).	6
Figure 5: The YAWL Worklist's web.xml file (detail).....	6
Figure 6: The Worklet Rules Editor	7
Figure 7: Example Top-level Specification	9
Figure 8: Associating a task with the Worklet Service	9
Figure 9: The TreatFever Worklet.....	10
Figure 10: Net-level Variables for the TreatFever Specification.....	11
Figure 11: Example Handler Process in the Rules Editor.....	17
Figure 12: Excerpt of Rule Set file Casualty_Treatment.xrs	18
Figure 13: Example Rule Tree (Casualty Treatment spec).....	19
Figure 14: Rules Editor First Time Use Message	21
Figure 15: The Rules Editor Configuration Dialog.....	22
Figure 16: Rules Editor Main Screen	23
Figure 17: Add New Rule Form.....	26
Figure 18: The Choose Worklet dialog	28
Figure 19: The New Rule Node Panel after a New Rule has been Defined.....	29
Figure 20: Message Dialog Offering to Replace the Running Worklet	29
Figure 21: Result of Replace Request Dialog.....	30
Figure 22: Main Screen after Addition of New Rule	30
Figure 23: The Specification Location Dialog.....	31
Figure 24: The Create New Rule Set Screen	32
Figure 25: Creating a New Rule Tree.....	33
Figure 26: The Draw Conclusion Dialog.....	35
Figure 27: A Conclusion Sequence shown as Text (detail).....	36
Figure 28: Welcome Page for the Worklet Service.....	37
Figure 29: Launching a Casualty Treatment Case (detail)	38
Figure 30: Editing the Admit Workitem (detail).....	38
Figure 31: Editing the Triage Workitem (detail)	39
Figure 32: New Case Launched by the Worklet Service.....	40
Figure 33: TreatFever Specification Uploaded and Launched.....	41

Figure 34: Test Fever Workitem (detail)	41
Figure 35: Treat Fever Workitem (detail).....	42
Figure 36: Discharge Workitem with Data Mapped from TreatFever Worklet.....	42
Figure 37: The wListMaker Specification	43
Figure 38: Start of wListMaker Case with Three ‘Bob’ Values Entered (detail)	43
Figure 39: Workitems from each of the 3 Launched Worklet Cases	44
Figure 40: ‘Bob’ Specifications Loaded and Launched by the Worklet Service.....	45
Figure 41: The Show List Workitem Showing the Changes to the Data Values	45
Figure 42: The OrganiseConcert Specification	46
Figure 43: The Book Stadium Workitem (detail)	47
Figure 44: The Sell Tickets Workitem (detail)	47
Figure 45: Effective Composite Rule for Do Show’s Pre-Item Constraint Tree	48
Figure 46: The Book Ent Centre Workitem (detail).....	48
Figure 47: Workflow Specifications Screen, OrganiseConcert case running	49
Figure 48: Case Viewer Screen	50
Figure 49: Raise Case-Level Exception Screen (Organise Concert example)	50
Figure 50: Available Work Items after External Exception Raised	51
Figure 51: The Timeout Test 3 Specification	52
Figure 52: Rules Editor Showing Single Timeout Rule for Receive_Payment Task	53
Figure 53: Rule detail for Receive Reply	53
Figure 54: Available Work After CancelOrder Worklet Launched.....	54
Figure 55: Reject Worklet Selection Screen.....	55
Figure 56: Example of a New Case-Level Exception Definition	56
Figure 57: Administration Tasks Screen (detail)	56

1. Welcome to the Worklet Service for YAWL

What is a Custom YAWL Service?

An important point of extensibility of the YAWL system is its support for interconnecting external applications and services with the workflow execution engine using a service-oriented approach. This enables running workflow instances and external applications to interact with each other in order to delegate work, to signal the creation of process instances and workitems, or to notify a certain event or a change of status of existing workitems.

Custom YAWL services are external applications that interact with the YAWL engine through XML/HTTP messages via certain endpoints, some located on the YAWL engine side and others on the service side. Custom YAWL services are registered with the YAWL engine by specifying their location, in the form of a “base URL”. Once registered, a custom service may send and receive XML messages to and from the engine.

More specifically, Custom YAWL services are able to check-out and check-in workitems from the YAWL engine. They receive a message when an item is enabled, and therefore can be checked out. When the Custom YAWL service is finished with the item it can check it back in, in which case the engine will set the work item to be completed, and proceed with the execution.

What is the YAWL Worklet Service?

The *Worklet Dynamic Process Selection & Exception Handling Service* for YAWL comprises two distinct but complementary services: a Selection Service, which enables dynamic flexibility for YAWL process instances; and an Exception Handling Service, which provides facilities to handle both expected and unexpected process exceptions (i.e. events and occurrences that may happen during the life of a process instance that are not explicitly modelled within the process) at runtime. A brief introduction to each Service follows.

The Selection Service

The Worklet Dynamic Process Selection Service (or *Selection Service*) enables flexibility by providing a process designer with the ability to substitute a workitem in a YAWL process at runtime with a dynamically selected “worklet” - a discrete YAWL process that acts as a sub-net for the workitem and so handles one specific task in a larger, composite process activity. The worklet is dynamically selected and invoked and may be created at any time, unlike a static sub-process that must be defined at the same time as, and remains a static part of, the main process model.

An extensible repertoire (or catalogue) of worklets is maintained by the Service. Each time the Service is invoked for a workitem, a choice is made from the repertoire based on the contextual data values within the workitem, using an extensible set of rules to determine the most appropriate substitution.

The workitem is checked out of the YAWL engine, and then the selected worklet is launched as a separate case. The data inputs of the original workitem are mapped to the inputs of the worklet. When the worklet has completed, its output data is mapped back to the original workitem, which is then checked back into the engine, allowing the original process to continue.

Worklets can be substituted for atomic tasks and multiple-instance atomic tasks. In the case of multiple-instance tasks, a worklet is launched for each child workitem. Because each child workitem may contain different data, the worklets that substitute for them are individually selected, and so may all be different.

The repertoire of worklets can be added to at any time, as can the rules base used for the selection process. Thus the service provides for dynamic ad-hoc change and process evolution, without having to resort to off-system intervention and/or system downtime, or modification of the original process specification.

The Exception Service

During every instance of a workflow process, certain things happen “off-plan”. That is, it doesn’t matter how much detail has been built into the process model, certain events occur during execution that affect the work being carried out, but were not defined as part of the process model. Typically, these events are handled “off-system” so that processing may continue. In some cases, the process model will be modified to capture this unforeseen event, which involves an organisational cost (downtime, remodelling, testing and so on).

The Worklet Dynamic Exception Handling Service (or *Exception Service*) provides the ability to handle these events in a number of ways and have the process continue unhindered. Additionally, once an unexpected exception is handled a certain way, that method automatically becomes an implicit part of the process specification for all future instances of the process, which provides for continuous evolution of the process but avoiding the need to modify the original process definition.

The Exception Service uses the same repertoire of worklets and dynamic rules approach as the Selection Service. The difference is that, while the Selection Service is invoked for certain tasks in a YAWL process, the Exception Service, when enabled, is invoked for every case and task executed by the YAWL engine, and will detect and handle up to ten different kinds of process exception. As part of the handling process, a process designer may choose from various actions (such as cancelling, suspending, restarting and so on) and apply them at a workitem, case and/or specification level.

The Exception Service is extremely flexible and multi-faceted, and allows a designer to provide tailor-made solutions to runtime process exceptions, as described in the following pages.

2. Installation

Worklet Installation Package

Within the YAWL web release *YAWL-Beta-8-Executable-WebServer-Version.zip* there is a file called **workletService.war** – this is the worklet web application component. In addition to the custom service application files, *workletService.war* contains a file called **workletRepository.zip**, which contains a required set of directories where worklets and rules files are stored, logs are written to and so on, and a number of sample worklets and rule sets (including all those discussed in this manual).

Installing the Worklet Service

1. Extracting and Installing the Repository

The repository needs to be installed first because the Service depends on its various components for its operation. The repository has this structure:

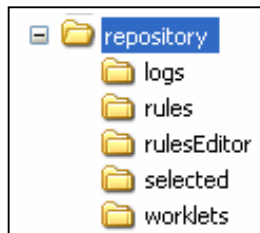


Figure 1: Worklet Repository Folder Structure

- § The *logs* folder is where the Service writes certain log files during its operation. In particular, the *eventLog.csv* file logs the key events of the service (when database persistence is disabled). All are plain text files.
- § The *rules* folder contains the sets of rules used during the selection and exception handling processes. Rules files have an XML format. Each YAWL specification that uses the service will have a corresponding rules file of the same name, except with an extension of *xrs* (XML rule set).
- § The *rulesEditor* folder contains a tool that is used to manage and modify rule sets (see *Chapter 5* for a complete description of the Rules Editor).
- § The *selected* folder contains a set of files, in XML format, that are essentially log files that capture the results of each selection process (either via the Selection Service, or through the selection of a compensating

worklet via the Exception Service). These files have an *xws* extension (XML Worklet Selection). These files are used by the Rules Editor to enable the addition of new rules for a specification.

- § The *worklets* folder contains the worklet specification files. These files are YAWL specifications that are run as required by the service.

Several of the folders contain examples.

8

To install the repository, first open *workletService.war* in an unzipping tool (e.g. 7-Zip). Locate and extract *workletRepository.zip* to a temporary directory.

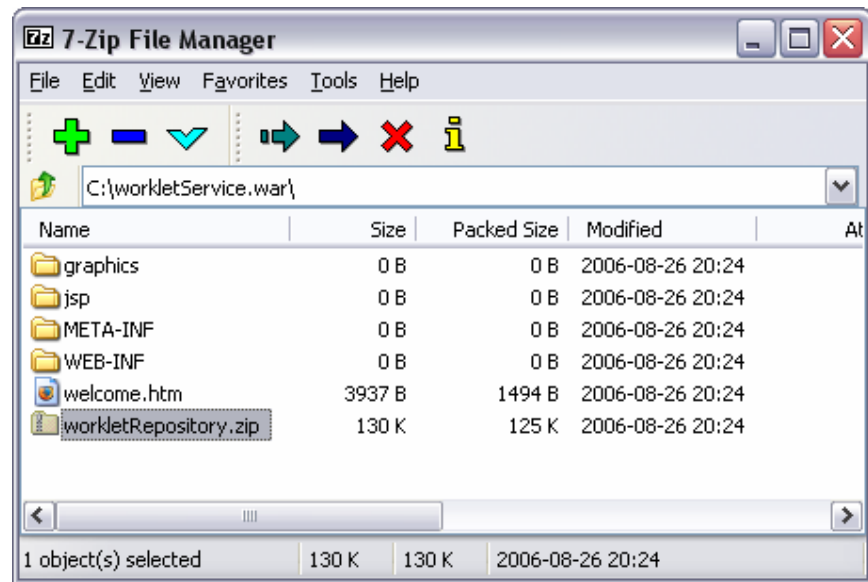


Figure 2: Contents of the workletService.war file

Then, simply extract the contents of *workletRepository.zip* to a folder of your choice, keeping the sub-folder structure intact.

Once that is done, the Service needs to be configured so that it can find the location of the repository.

2. Configuring the Service

The *workletService.war* file should be copied to the *webapps* directory of your Tomcat installation (if necessary, refer to the YAWL Installation Manual for more information). Then, the file needs to be extracted to its own directory under *webapps*. The easiest way to achieve this is to simply start Tomcat – it will automatically extract, install and start the Worklet Service.

Once the service is fully extracted and installed, there are a couple of minor configuration tasks to complete.

8

Open the service's *web.xml* file (in folder `\webapps\workletService\WEB-INF\`) in any text editor.

Locate the parameter named *Repository*, and change its *param-value* to the path where you installed the worklet repository. The value should include the folder 'repository' and end with a slash, as shown in Figure 3 below.

```
<context-param>
  <param-name>Repository</param-name>
  <param-value>C:\Worklets\repository\</param-value>
  <description>
    The path where the worklet repository is installed.
  </description>
</context-param>

<context-param>
  <param-name>EnablePersistence</param-name>
  <param-value>>false</param-value>
  <description>
    'true' to enable persistence and logging
    'false' to disable
  </description>
</context-param>
```

Figure 3: The Worklet Service's *web.xml* file (detail).

If you have enabled database persistence for the YAWL Engine (if necessary, see the YAWL Install Manual for details about enabling persistence for the YAWL Engine), then persistence should also be enabled for the Worklet Service (so that case data for running processes can be persisted across Tomcat sessions). To enable persistence, change the *param-value* to **true** for the *EnablePersistence* parameter.

Save and close *web.xml*. Worklet Service installation is now complete. However, by default the Selection Service is enabled within the YAWL Engine, but the Exception Service is not. If you wish to enable the Exception Service, proceed to Step 3 below.

3. Enabling the Worklet Exception Service

When YAWL Beta 8 is installed, the Exception Service is configured as disabled in the YAWL Engine. To enable the Exception Service, a parameter has to be set in the YAWL Engine's *web.xml* file.

8

Open the engine's *web.xml* file (in folder `\webapps\yaw\WEB-INF\`).

Locate the parameter named *EnableExceptionService*; to enable the Exception Service, change the *param-value* to **true** (see Figure 4). Save and close *web.xml*.

```

<!-- PARAMS FOR EXCEPTION SERVICE -->

<context-param>
  <param-name>EnableExceptionService</param-name>
  <param-value>true</param-value>
  <description>
    Set this value to 'true' to enable monitoring by an
    Exception Service (specified by the URI param below).
    Set it to 'false' to disable the Exception Service.
  </description>
</context-param>

```

Figure 4: The YAWL Engine's *web.xml* file (detail).

8

The Exception Service uses extensions (or 'hooks') in the YAWL worklist handler to provide methods for interacting with the Service, so if you have enabled the Service in the Engine as above, you also need to enable the extensions in the worklist. To do so, locate and open the worklist's *web.xml* file (in folder *\webapps\worklist\WEB-INF*).

Locate the context parameter named *InterfaceX_BackEnd*. By default, the entire parameter block is commented out. To enable the Exception Service extensions to the worklist, simply remove the comment tags (shown in red in Figure 5 below). Save and close *web.xml*.

```

<!-- This param, when available, enables the worklet exception
      service add-ins to the worklist. If the exception service
      is enabled in the engine, then this param should also be
      made available. If it is disabled in the engine, the
      entire param should be commented out. -->
<!--
<context-param>
  <param-name>InterfaceX_BackEnd</param-name>
  <param-value>http://localhost:8080/workletService</param-value>
  <description>
    The URL location of the worklet exception service.
  </description>
</context-param>
-->

```

Figure 5: The YAWL Worklist's *web.xml* file (detail).

The Exception Service is now fully enabled and operational. For information on how the Exception Service works and how to use it, see Sections 4 and 6.

4. Installing the Rules Editor

The Worklet Rules Editor allows you to browse the rule sets of specifications, add new rules to existing rule sets, and add complete, new rules trees to rule sets. It is a .NET based application, so has the following requirements:

§ Operating System: Windows OS's 98SE or better

- § The Microsoft .NET framework (any version). If you don't have the framework installed, it can be downloaded for free from Microsoft: <http://www.microsoft.com/downloads/>

The Rules Editor tool is found in the *rulesEditor* folder of the worklet repository. It can be executed directly from there – no further installation is required.

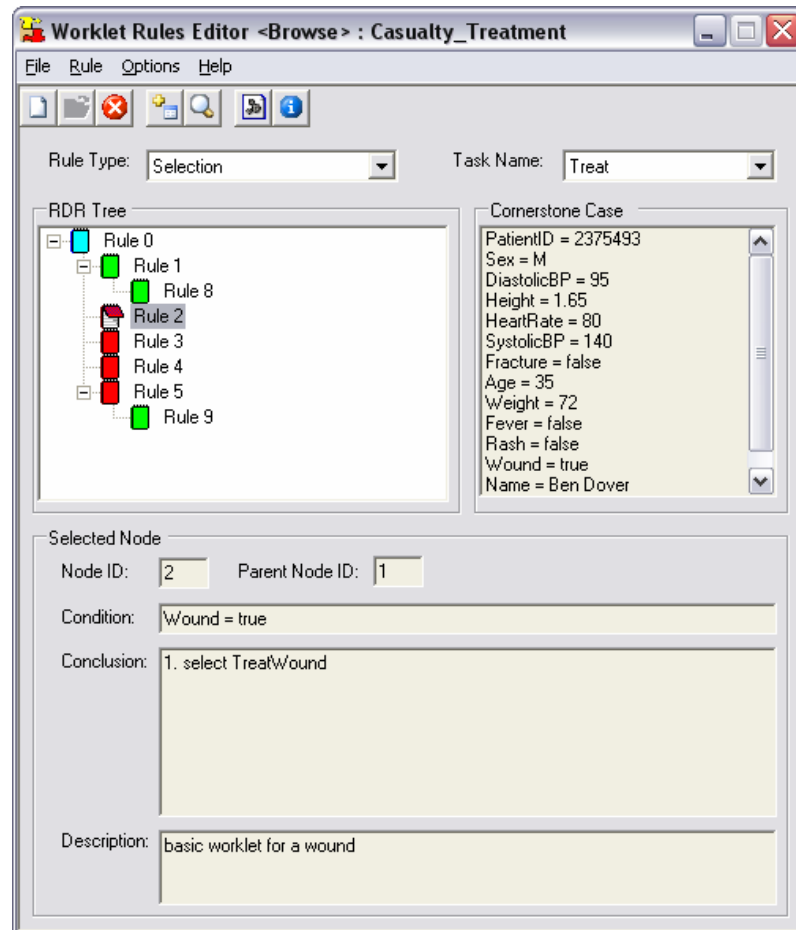


Figure 6: *The Worklet Rules Editor*

For more details on the Rules Editor and how to use it, refer to *Section 5*.

3. Using the Worklet Selection Service

Fundamentally, a worklet is nothing more than a workflow specification that has been designed to perform one part of larger or ‘parent’ specification. However, it differs from a decomposition or sub-net in that it is dynamically assigned to perform a particular task at runtime, while sub-nets are statically assigned at design time. Also, worklets can be added to the repertoire at any time during the life of a specification, even while instances are running. So, rather than being forced to define all possible “branches” in a specification when it is first defined, the Worklet Service allows you to define a much simpler specification that will evolve dynamically as more worklets are added to the repertoire for a particular task as different contexts arise.

The first thing you need to do to make use of the service is to create a number of YAWL specifications – one which will act as the top-level (or manager or parent) specification, and one or more worklets which will be dynamically substituted for particular top-level tasks at runtime.

The YAWL Editor is used to create both top-level and worklet specifications. A knowledge of creating and editing YAWL specifications, and the definition of data variables and parameters for tasks and specifications, is assumed. For more information on how to use the YAWL Editor, see the [YAWL Editor User Manual](#).

Before opening the YAWL Editor, make sure that the Worklet Service is correctly installed and that Tomcat is running (see Section 2 of this document and/or the [YAWL Engine Installation Manual](#) for more information).

First, a top-level specification needs to be defined.

Top-level or Parent Specifications

To define a top-level specification, open the YAWL Editor, and create a process specification in the usual manner. Choose one or more tasks in the specification that you want to have replaced with a worklet at runtime. Each of those tasks needs to be associated via the YAWL Editor with the Worklet Service.

For example, Figure 7 shows a simple specification for a Casualty Treatment process. In this process, we want the *Treat* task to be substituted at runtime with the appropriate worklet based on the patient data collected in the *Admit* and *Triage* tasks. That is, depending on each patient’s actual physical data and reported symptoms, we would like to run the worklet that best handles the patient’s condition.

Worklets may be associated with an atomic task, or a multiple-instance atomic task. Any number of worklets can be associated with (i.e. comprise the repertoire of) an individual task, and any number of tasks in a particular specification can be associated with the Worklet Service.

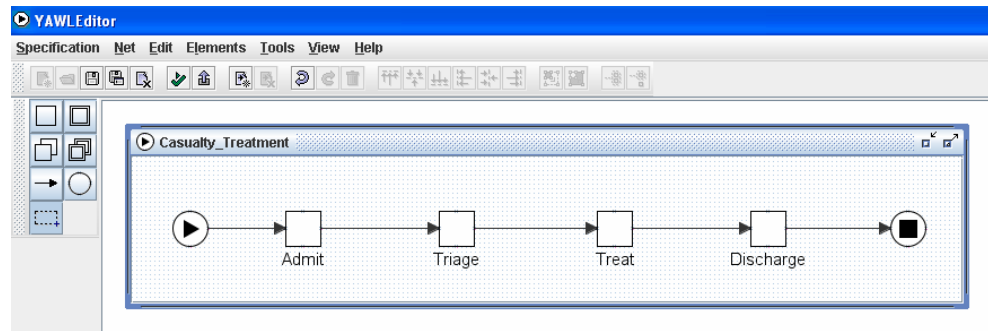


Figure 7: Example Top-level Specification

8

Here, we want to associate the *Treat* task with the Worklet Service. To do so, right click on the task, then select *Task Decomposition Detail* from the popup menu. The *Update Task Decomposition* dialog is shown (Figure 8). This dialog shows the variables defined for the task – each one of these maps to a net-level variable, so that in this example all of the data collected from a patient in the first two tasks are made available to this task. The result is that all of the relevant current case data for this process instance can be used by the Worklet Service to enable a contextual decision to be made. Note that it is not necessary to map all available case data to a worklet enabled task, only that data required by the Service to make an appropriate decision. How this data is used will be discussed later in this manual.

Name	Type	Usage
PatientID	string	Input Only
Age	long	Input Only
Sex	string	Input Only
HeartRate	long	Input Only
SystolicBP	long	Input Only
DiastolicBP	long	Input Only
Height	double	Input Only
Weight	double	Input Only
Fracture	boolean	Input Only
Fever	boolean	Input Only
Rash	boolean	Input Only
AbdominalPain	boolean	Input Only
Wound	boolean	Input Only
Pharmacy	string	Input & Output
Treatment	string	Input & Output
Notes	string	Input & Output

YAWL Registered Service Detail

YAWL Service: Worklet Dynamic Process Selection Service

- Default Engine Worklist
- RPC-Style Web Service Invoker
- SMS Message Module. Works if you have an account.
- Time service, allows tasks to be a timeout task.
- Worklet Dynamic Process Selection Service

Figure 8: Associating a task with the Worklet Service

The list of task variables in Figure 8 also show that most variables are defined as ‘Input Only’ – this is because those values will not be changed by any of the worklets that may be executed for this task; they will only be used in the selection process. The last three variables are defined as ‘Input & Output’, so that the worklet can “return”, or map back to these variables, data values that are captured during the worklet’s execution.

The dialog has a section at the bottom called *YAWL Registered Service Detail*. It is here that the task is associated with the Worklet Service by choosing the Worklet Service from the list of available services. Note that list of services will only be seen if Tomcat is currently running and it has those services installed.

8

Select the Worklet Service from the list. That is all that’s required to make the top-level specification worklet-enabled. Next, we need to create one or more worklet specifications to execute as substitutes for the worklet-enabled task.

Worklet Specifications

When the *Casualty Treatment* top-level specification is executed, the YAWL Engine will notify the Worklet Service when the worklet-enabled *Treat* task becomes enabled. The Worklet Service will then examine the data in the task and use it to determine which worklet to execute as a substitute for the task. Any or all of the data in the task may also be mapped to the selected worklet case as input data. Once the worklet instance has completed, any or all of the available output data of the worklet case may be mapped back to the *Treat* task to become its output data, and the top-level process will continue.

A worklet specification is a standard YAWL process specification, and as such is created in the YAWL Editor in the usual manner. Each of the data variables that are required to be passed from the parent task to the worklet specification need to be defined as net-level variables in the worklet specification.

Figure 9 shows a simple example worklet to be substituted for the *Treat* top-level task when a patient complains of a fever.

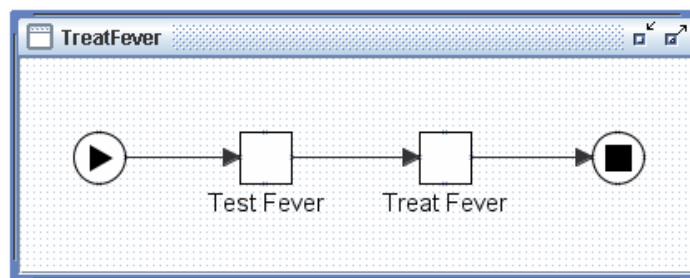


Figure 9: The TreatFever Worklet

In itself, there is nothing special about the *TreatFever* specification. Even though it will be considered by the Worklet Service as a member of the worklet repertoire and may thus be considered a “worklet”, it is a standard YAWL specification and

as such may be executed directly by the YAWL engine without any reference to the Worklet Service.

As mentioned previously, those data values that are required to be mapped from the parent task to the worklet need to be defined as net-level variables in the worklet specification. Figure 10 shows the net-level variables for the *TreatFever* task.

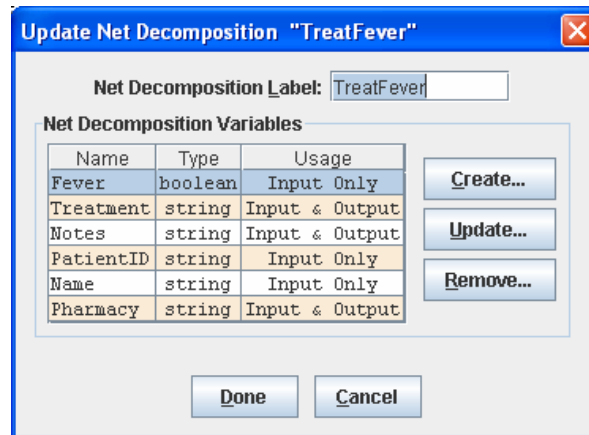


Figure 10: Net-level Variables for the *TreatFever* Specification

Note the following:

- § Only a sub-set of the variables defined in the parent *Treat* task (see Figure 8) are defined here. It is only necessary to map from the parent task those variables that contain values to be displayed to the user, and/or those variables that the user will supply values for to be passed back to the parent task when the worklet completes.
- § The definition of variables is not restricted to those defined in the parent task. Any additional variables required for the operation of the worklet may also be defined here.
- § Only those variables that have been defined with an identical name and data type to variables in the parent task and with a *Usage* of 'Input Only' or 'Input & Output' will have data passed into them from the parent task when the worklet is launched.
- § Only those variables that have been defined with an identical name and data type to variables in the parent task and with a *Usage* of 'Output Only' or 'Input & Output' will pass their data values back to the parent task when the worklet completes.

In Figure 10, it can be seen that the values for the *PatientID*, *Name* and *Fever* variables will be used by the *TreatFever* worklet as display-only values; the *Notes*, *Pharmacy* and *Treatment* variables will receive values during the execution of the worklet and will map those values back to the top-level *Treat* task when the worklet completes.

The association of tasks with the Worklet Service is not restricted to top-level specifications. Worklet specifications also may contain tasks that are associated with the Worklet Service and so may have worklets substituted for them, so that a hierarchy of executing worklets may sometimes exist. It is also possible to recursively define worklet substitutions – that is, a worklet may contain a task that, while certain conditions hold true, is substituted by another instance of the same worklet specification that contains the task.

Any number of worklets can be created for a particular task. For the *Casualty Treatment* example, there are (initially) five worklets in the repertoire for the *Treat* task, one for each of the five primary conditions that a patient may present with in the *Triage* task: Fever, Rash, Fracture, Wound and Abdominal Pain. Which worklet is chosen for the *Treat* task depends on which of the five is given a value of *True* in the *Triage* task.

How the Worklet Service uses case data to determine the appropriate worklet to execute is described in Section 5.

4. Using the Worklet Exception Service

In the previous Section, we saw how the Worklet Service adds dynamic flexibility to a usually static YAWL specification by substituting tasks with contextually chosen worklets at runtime. The Worklet Exception Service leverages off the worklet framework to also provide support for the myriad exceptions that may occur during the execution of any process instance.

Every process instance, no matter how rigidly structured, will experience some kind of exception during its execution. While the word ‘exception’ conjures up ideas of errors or problems occurring within the executing process instance, the meaning in terms of workflow processes is much broader: exceptions are merely events or occurrences that, for one reason or another, were not defined in the process model. It may be that these events are known to occur in a small number of cases, but not enough to warrant their inclusion in the process model; or they may be things that were never expected to occur (or may be never even imagined *could* occur). In any case, when they do happen, if they are not part of the process model, they must either be handled “off-line” before the process continues (and the way they are handled is rarely recorded) or in some instances the entire process must be aborted.

Alternately, an attempt might be made to include every possible twist and turn into the process model so that when such events occur, there is a branch in the process to take care of it. This approach may lead to very complex models where much of the original business logic is obscured, and doesn’t avoid the same problems when the next unexpected exception occurs.

The Exception Service addresses these problems by allowing you to define worklets that will act as exception handling processes for parent workflow instances when certain events occur. Rules are defined in much the same way as for the Selection Service, but with added features that enable you to pause, resume, cancel or restart the task, case, or all cases of a specification, that triggered the exception.

Because the service allows you to define exception handlers for all exception events, and even to add new handlers at runtime, all exception events are able to be captured “on-system”, so that the handlers are available to all future occurrences of a particular event for the same context. And, since the handlers are worklets, the original parent process model only needs to contain the actual business logic for the process, while the repertoire of handlers grows as new exceptions arise or different ways of handling exceptions are formulated.

IMPORTANT: While the Selection Service is linked explicitly to tasks as defined in the YAWL Editor, and thus available whenever a worklet-enabled task is executed, the Exception Service is either enabled or disabled (on or off); when it is enabled, it manages exception handling for **all** process instances executed by the engine –explicitly linking a task or process to the service is not required. Also, the Selection and Exception Services can be used in combination within particular case instances to achieve dynamic flexibility and exception handling simultaneously.

Exception Types

This section introduces the 10 different types of exception that have been identified, seven of which are supported by this version of the Worklet Service. Some are related, while others are more distinct. Later sections will show examples of each of these.

When the Exception Service is enabled, it is notified whenever any of these exception types occur for every process instance executed by the YAWL Engine (by various means). The Exception Service maintains a set of rules (described in detail in Section 5) that are used to determine which exception handling process, if any, to invoke. If there are no rules defined for a certain exception type for a specification, the exception event is simply ignored by the service. Thus you only need to define rules for those exception events that you actually want to handle for a particular specification.

Constraint Types

Constraints are rules which are applied to a workitem or case immediately before or after execution of that workitem or case begins. Thus, there are four types of constraint exception:

- § **CasePreConstraint** – case-level pre-constraint rules are checked when each case (i.e. instance) begins execution;
- § **ItemPreConstraint** – item-level pre-constraint rules are checked when each workitem in a case becomes enabled (i.e. ready to be checked out);
- § **ItemPostConstraint** – item-level post-constraint rules are checked when each workitem moves to a completed status; and
- § **CasePostConstraint** – case-level post constraint rules are checked when a case completes.

The Exception Service receives notification from the YAWL Engine when each of these events occur, then checks the rule set for the case to determine, firstly, if there are any rules of that type defined for the case, and if so, if any of the rules evaluate to true using the contextual data of the case or workitem. If the rule set finds a matching rule for the exception type and data, an exception process is invoked.

Note that for each of the constraint events, an exception process is invoked for a rule when that rule's condition evaluates to **true**. So, for example, if the condition of an ItemPreConstraint rule for a *Triage* task was "PrivateInsurance=false", and that value of that attribute in the workitem was also false, then the exception process for that rule would be invoked.

Externally Triggered Types

Externally triggered exceptions occur, not through the case's data values, but because something has happened outside of the process execution that has an affect on the continuing execution of the process. Thus, these events are triggered by a user; depending on the actual event, a particular handler will be invoked.

There are two types of external exceptions, **CaseExternalTrigger** (for case-level

events) and **ItemExternalTrigger** (for item-level events). See later in this section for examples of each and how they are invoked.

TimeOut

A timeout event occurs when a workitem is linked to the YAWL Time Service and the deadline set for that workitem is reached. In this case, the YAWL Engine notifies the Exception Service of the timeout event, and passes to the service a reference to itself and each of the other workitems that were running in parallel with the timed-out item. Therefore, timeout rules may be defined for each of the workitems affected by the timeout (including the actual timeout workitem itself).

ItemAbort

An ItemAbort event occurs when a workitem being handled by an external program (as opposed to a human user) reports that the program has aborted before completion. This event is not supported by this version of Exception Service.

ResourceUnavailable

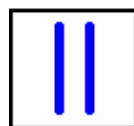
This event occurs when an attempt has been made to allocate a workitem to a resource and the resource reports that it is unavailable to accept the allocation. This event is not currently supported by the Exception Service.

ConstraintViolation

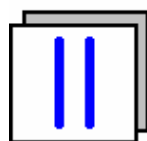
This event occurs when a data constraint has been violated for a workitem *during execution* (as opposed to pre or post execution). This event is not currently supported by the Exception Service.

Exception Handling Primitives

For any exception event that occurs, a handling process may be invoked. Each handling process contains a number of steps, or *primitives*, in sequence, and is defined graphically using the Worklet Rules Editor (see Section 5). Each of the handling primitives is introduced below.



Suspend WorkItem – suspends (or pauses) execution of a workitem, until it is continued, restarted, cancelled, failed or completed, or its parent case is cancelled or completed.



Suspend Case – suspends all “live” workitems in the current case instance (a live workitem has a status of fired, enabled or executing), effectively suspending execution of the case.



Suspend All Cases – suspends all “live” workitems in all of the currently executing instances of the specification in which the workitem is defined, effectively suspending all running cases of the specification.



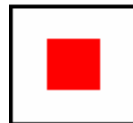
Continue Workitem – un-suspends (or continues) execution of the previously suspended workitem.



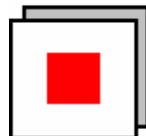
Continue Case – un-suspends execution of all previously suspended workitems for the case, effectively continuing case execution.



Continue All Cases – un-suspends execution of all previously suspended workitems for all cases of the specification in which the workitem is defined or of which the case is an instance, effectively continuing all running cases of the specification.



Remove Workitem – removes (or cancels) the workitem; execution ends, and the workitem is marked with a status of cancelled. No further execution occurs on the process path that contains the workitem.



Remove Case – removes (cancels) the case. Case execution ends.



Remove All Cases – removes (cancels) all case instances for the specification in which the workitem is defined, or of which the case is an instance.



Restart Workitem – rewinds workitem execution back to start. Resets the workitem’s data values to those it had when it began execution.



Force Complete WorkItem – completes a “live” workitem. Execution of the workitem ends, and the workitem is marked with a status of ForcedComplete, which is regarded as a successful completion, rather than a cancellation or failure. Execution proceeds to the next workitem on the process path.



Force Fail Workitem – fails a “live” workitem. Execution of the workitem ends, and the workitem is marked with a status of Failed, which is regarded as an unsuccessful completion, but not a cancellation – execution proceeds to the next workitem on the process path.



Compensate – run a compensatory process (i.e. a worklet). Depending on previous primitives, the worklet may execute simultaneously to the parent case, or execute while the parent is suspended (or even removed).

The primitives “Suspend All Cases”, “Continue All Cases” and “Remove All Cases” may be edited so that their action is restricted to ancestor cases only. Ancestor cases are those in a hierarchy of worklets back to the parent case (that is, where a case invokes a worklet which invokes another worklet and so on).

An example of a definition of an exception handling process in the Rules Editor is below:

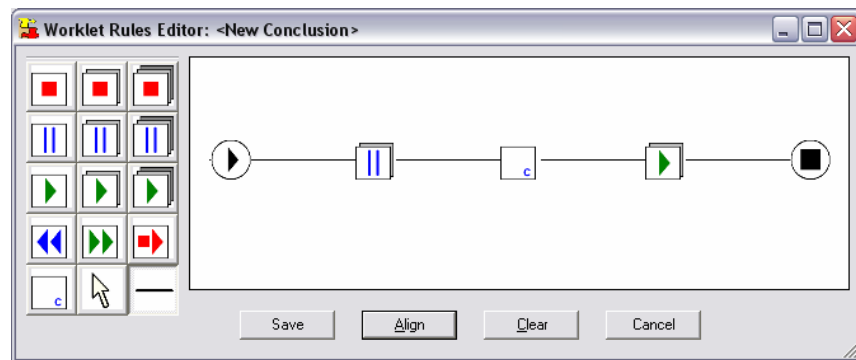


Figure 11: Example Handler Process in the Rules Editor

When invoked, this handler will suspend the current case, then run a compensating worklet, then continue execution of the case.

In the same manner as the Selection Service, the Exception Service also supports data mapping from a case to a compensatory worklet and back again. For example, if a certain variable has a value that prevents a case instance from continuing, a worklet can be run as a compensation, during which a new value can be assigned to the variable and that new value mapped back to the parent case, so that it may continue execution.

The full capabilities of the Exception Service are better described in the walkthroughs in Section 6. But before we consider the walkthroughs, we must first look at exactly how the rule sets are formed and how they operate, and how to use the Worklet Rules Editor to manage rule sets for specifications. These topics are discussed in the next section.

5. Worklet Rule Sets and the Rules Editor

This section describes the structure and operation of worklet rule sets. A tool has been designed to manage the creation and modification of rule sets for specifications, called the *Worklet Rules Editor*. The structure and operation of rule sets is best described by using the Rules Editor to display and manipulate them. So a description of how to use the Rules Editor is interspersed in this section with the description of the rule sets themselves.

Again, the Worklet Selection and Exception Services work in very similar ways, but with some necessary differences. In this section, the discussion of rule sets applies to both services, except where indicated.

Worklet Rule Sets

Any YAWL specification may have an associated rule set. The rule set for each specification is stored as XML data in a disk file that has the same name as the specification, but with an “.xrs” extension (XML Rule Set). All rule set files are stored in the *rules* folder of the worklet repository. For example, the file *Casualty_Treatment.xrs* contains the worklet rule set for the *Casualty_Treatment.xml* YAWL process specification. Figure 12 shows an excerpt from that file.

```
        </ruleNode>
    </pre>
</case>
</constraints>
<selection>
  <task name="Treat">
    <ruleNode>
      <id>0</id>
      <parent>-1</parent>
      <trueChild>1</trueChild>
      <falseChild>-1</falseChild>
      <condition>True</condition>
      <conclusion>null</conclusion>
      <cornerstone> </cornerstone>
      <description>root level default node</description>
    </ruleNode>
    <ruleNode>
      <id>1</id>
      <parent>0</parent>
      <trueChild>8</trueChild>
      <falseChild>2</falseChild>
      <condition>Fever = true</condition>
      <conclusion>
        <_1>
          <action>select</action>
          <target>TreatFever</target>
        </_1>
      </conclusion>
      <cornerstone>
```

Figure 12: Excerpt of Rule Set file *Casualty_Treatment.xrs*

A rule set for a specification consists of a collection of *rule trees*. Each rule tree represents a set of modified *Ripple-Down Rules (RDR)*, which maintains a rule node hierarchy in a binary-tree structure. When a rule tree is queried, it is traversed from the root node of the tree along the branches, each node having its condition evaluated along the way. If a node's condition evaluates to *True*, and it has a true child, then that child node's condition is also evaluated. If a node's condition evaluates to *False*, and there is a false child, then that child node's condition is evaluated. When a terminal node is reached (i.e. a node without any child nodes) if its condition evaluates to *True*, then that conclusion is returned as the result of the tree traversal; if it evaluates to *False*, then the last node in the traversal that evaluated to *True* is returned as the result. The root node (Rule 0) of the tree is always a default node with a default *True* condition and conclusion, and so can only have a true branch.

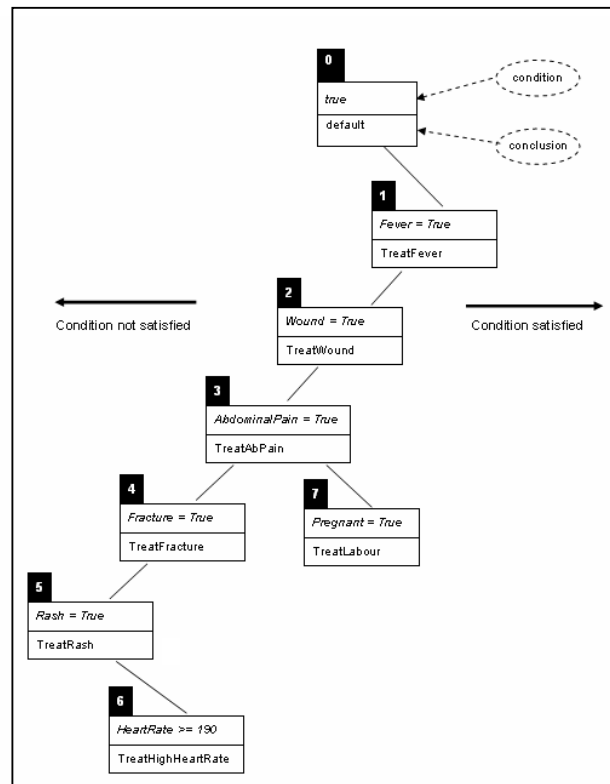


Figure 13: Example Rule Tree (*Casualty Treatment spec*)

Effectively, each rule node on the true branch of its parent node is an exception rule to the more general one of its parent (that is, a *refinement* of the parent rule), while each rule node on the false branch of its parent node is an “else” rule to its parent (or an alternate to the parent rule). For example, see the selection rule tree for the *Casualty Treatment* specification (Figure 13). The condition part is the rule that is evaluated, and the conclusion is the name of the worklet selected by that rule if the condition evaluates to true. For example, if the condition “fever = true”

evaluates to true, then the *TreatFever* worklet is selected (via node 1); if it is false, then the next false node is tested (node 2). If node 2 is also false, then node 3 is tested. If node 3 evaluates to true, then the *TreatAbPain* worklet is selected, **except if** the condition in its next true node (node 7) also evaluates to true, in which case the *TreatLabour* worklet is selected.

One worklet rule set file is associated with each specification, and may contain up to eleven sets of rule trees (or *tree sets*), one for selection rules and one for each of the ten exception types. Three of the eleven relate to case-level exceptions (i.e. CasePreConstraint, CasePostConstraint and CaseExternalTrigger) and so each of these will have only one rule tree in the tree set. The other eight tree sets relate to the workitem-level (seven exception types plus selection), and so may have one rule tree for each workitem in the specification – that is, the tree sets for these eight rule types may consist of a number of rule trees.

It is not necessary to define rules for all eleven types for each specification. You only need to define rules for those types that you want to handle – any exception types that aren't defined in the rule set file are simply ignored. So, for example, if you are only interested in capturing pre and post constraints at the workitem level, then only the ItemPreConstraint and ItemPostConstraint tree sets need to be defined (i.e. rules defined within those tree sets). In this example, any Timeout exception events that occur during the execution of the specification would be ignored by the Exception Service. Of course, rules for a Timeout event could be added later if required (as could any of the other types not yet defined in the rule set).

Referring back to Figure 12, notice that the file specifies a *Selection* rule tree for the *Treat* task. The second *ruleNode* contains a condition “Fever = True” and a conclusion of “TreatFever”. Thus, when the condition “Fever = True” evaluates to true, the worklet *TreatFever* is chosen as a substitute for the *Treat* task. Notice also that each rule node (except the first) has a parent, and may have two child nodes, a *true* child and a *false* child.

To summarise the hierarchy of a rule set (from the bottom up):

- § **Rule Node:** contains the details (condition, conclusion, id, parent and so on) of one discrete ripple-down rule.
- § **Rule Tree:** consists of a number of rule nodes in a binary tree structure.
- § **Tree Set:** a set of one or more rule trees. Each tree set is specific to a particular rule type (timeout, selection, etc.). The tree set of a case-level exception rule type will contain exactly one tree. The tree set of an item-level rule type will contain one rule tree for each task of the specification that has rules defined for it (not all tasks in the specification need to have a rule tree defined).
- § **Rule Set:** a set of one or more tree sets representing the entire set of rules defined for a specification. Each rule set is specific to a particular specification. A rule set will contain one or more tree sets – one for each rule type for which rules have been defined.

Of course, to maintain a rule set of any complexity by directly editing the XML in a rule set file would be daunting, to say the least. To make things much easier, a

Rules Editor tool has been developed, and can be found in the *rulesEditor* folder of the worklet repository. It can be run directly from there – no further installation is required (depending on the requirements below).

The Rules Editor

The Worklet Rules Editor allows for the addition of new rules to existing rule sets of specifications, and the creation of new rule sets. It is a .NET based application, so has the following requirements:

- § Operating System: Windows 98SE or later.
- § The Microsoft .NET framework (any version). If you don't have the framework installed, it can be downloaded free from Microsoft.

8

When the Rules Editor is run for the first time, the following dialog is displayed:

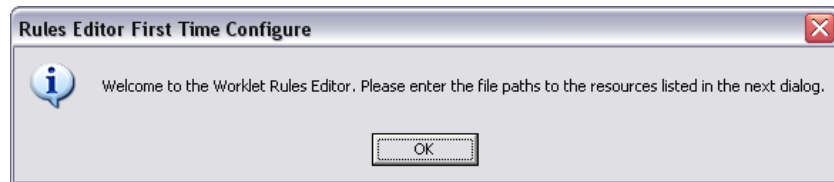


Figure 14: Rules Editor First Time Use Message

Clicking OK shows the Configuration dialog (Figure 15), where the paths to resources the Rules Editor uses are to be specified. Some default paths are shown, but can be modified directly or by using the browse buttons where available. The following paths must be specified:

- § **Worklet Service URI:** the URI to the Worklet Service. The default URI assumes it is installed locally. If it is remote to the computer running the Rules Editor, then that URI should be entered, ensuring it ends with “/workletService”.
- § **Worklet Repository:** the path where the worklet repository was installed. The default path shown assumes the Rules Editor was started from the *rulesEditor* folder of the repository. If it was started from another location, specify the actual path to the repository by editing the path or browsing to the correct location.
- § **Specification Paths:** the path or paths to locations on the local computer where YAWL specification files may be found. The rules editor will search each of the paths provided for specifications for which rule sets may be created. Multiple paths can be provided, separated by semicolons ‘;’.
- § **YAWL Editor:** the path and filename for the YAWL Editor.

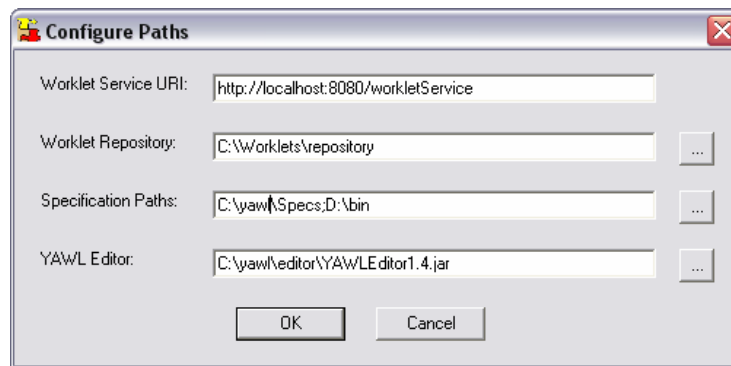


Figure 15: The Rules Editor Configuration Dialog

Some checks will take place to make sure the paths are valid and you will be asked to correct any that are not. Once the configuration is complete, the main screen will appear. This screen allows you to view each node of each rule tree defined for a particular specification. From this screen you can also add new rules to the current rule set, create new tree sets for an existing rule set, and create entirely new rule sets for new specifications.

Browsing an Existing Rule Set

8

To load a rule set into the Rules Editor, click on the *File* menu, then select *Open...*, or click on the *Open* toolbar button. The File Open Dialog will open with the *rules* folder of the repository selected. Select the file you wish to open, and then click OK.

Figure 16 shows the main screen with the rule set for the *Casualty Treatment* specification loaded. On this screen, you may browse through each node of a rule tree set and view the various parts of each node. The main features of the screen are explained below.

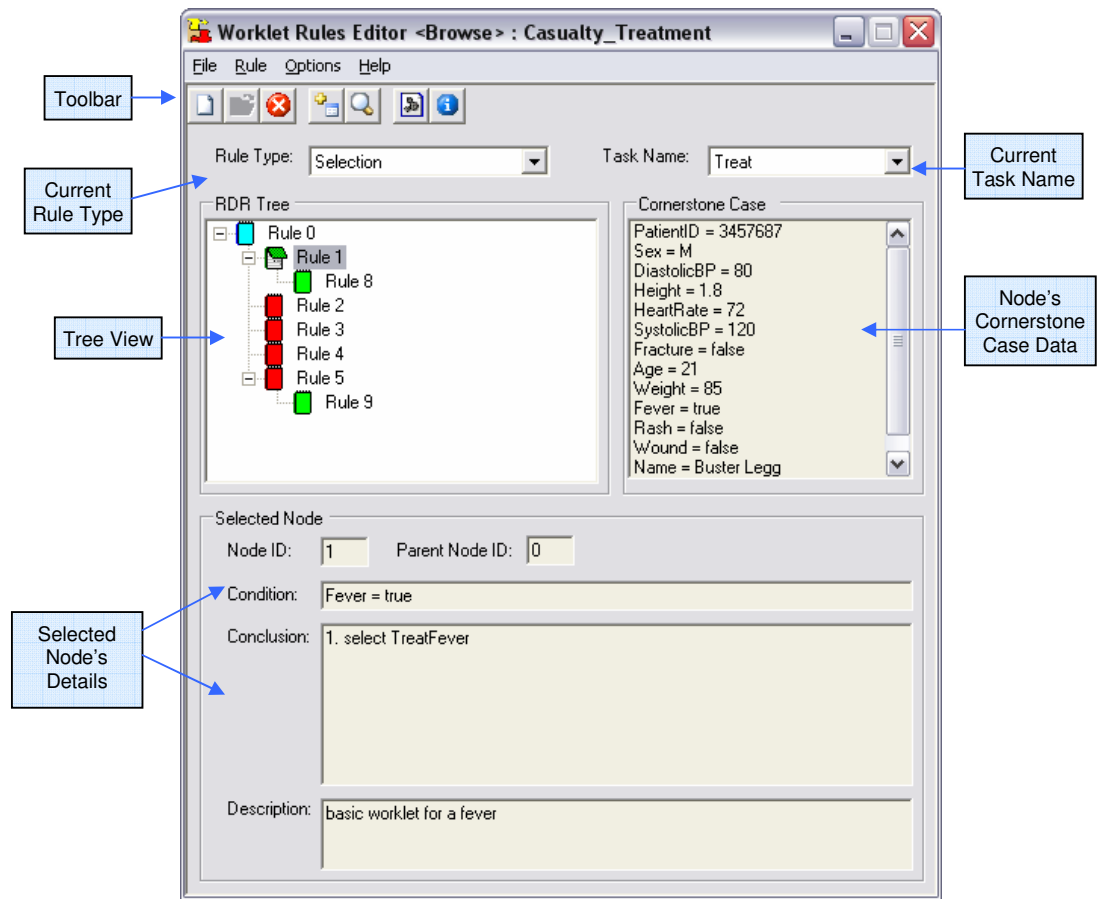
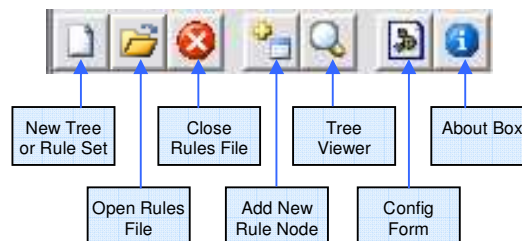


Figure 16: Rules Editor Main Screen

The Toolbar

The toolbar buttons replicate the functions available from the main menu.



- § **New Tree or Rule Set:** If there is no rules file currently open in the Editor, this button displays the *New Rule* form to allow the creation of an entirely new rule set file for a specification (i.e. one that does not yet have a rule set file defined). If there is a rules file currently open in the Editor, displays the *New Rule* form to allow the addition of new tree sets to the opened rule set file (for rule types that have not yet been defined for that specification). See the *Creating a New Rule Set and/or Tree Set* section below for more details.

- § **Open Rules File:** Opens an existing rules file for browsing and/or editing. The title bar shows the name of the specification associated with the currently loaded rule set.
- § **Close Rules File:** Closes the currently opened rules file. Only one rules file may be open at any one time.
- § **Add New Rule Node:** Displays the *Add Rule* form to allow the addition of a new rule node to an existing tree to refine a worklet selection. See the section below on the *Adding a New Rule* for more details.
- § **Tree Viewer:** Displays the *Tree Viewer* form, which provides the ability to view large trees in full-screen mode.
- § **Config Form:** Displays the configuration form discussed above.
- § **About Box:** Displays some information about the rules editor (version number, date and so on).

Other Features

- § **Current Rule Type:** This drop-down list displays each rule type that has a tree set defined in the opened rules file. Selecting a rule type from the list displays in the *Tree View* an associated rules tree from the tree set. Works in conjunction with the *Task Name* drop-down list.
- § **Current Task Name:** This drop-down list displays the name of each task that has a rules tree associated with it for the rule type selected in the *Rule Type* list. Selecting a task name will display the rules tree for that task in the *Tree View*. This drop-down list is disabled for case level rules types.
- § **Tree View:** This area displays the currently selected rules tree in a graphical tree structure. Selecting a node in the tree will display the details of that node in the *Selected Node* and *Cornerstone Case* panels. Nodes are colour coded for easier identification:
 - **Blue** nodes represent the root node of the tree
 - **Green** nodes are **true** or **exception** nodes (i.e. they are on a true branch from their parent node)
 - **Red** nodes are **false** or **else** nodes (i.e. they are on a false branch from their parent node)
- § **Selected Node:** Displays the details of the node currently selected in the *Tree View*.
- § **Cornerstone Case:** displays the complete set of case data that, in effect, caused the creation of the currently selected rule (see *Adding a new rule* below for more details). In Figure 16, the Cornerstone Case data shows that, amongst other things, the variable *Fever* had a value of true, while the variables *Rash*, *Wound* and *Fracture* each have value of false.

Adding a New Rule

There are occasions when the worklet returned for a particular case, while the correct choice based on the current rule set, is an inappropriate choice for the case. For example, if a patient in a *Casualty Treatment* case presents with a rash *and* a heart rate of 190, while the current rule set correctly returns the *TreatRash* worklet, it may be desirable to treat the racing heart rate before the rash is attended to. In such a case, as the Worklet Service begins execution of an instance of the *TreatRash* process, it is obvious that a new rule needs to be added to the rule set so that cases that have such data (both now and in the future) will be handled correctly.

8

To add a new rule to a particular tree of a rule set, it is first necessary to open the rule set in the Rules Editor (as described above). Then, click *Rules* on the top menu, then *Add...*, or click the *Add Rule* toolbar button, to open the (initially blank) *Add Rule* form.

Notice that the name of the opened rule set is shown in the title bar of the form, and the rule type and task name that are currently selected on the main form have been transferred to the *Add Rule* form. Thus, to add a new rule to a rule tree, that rule tree must first be selected on the main Rules Editor form before the *Add Rule* form is opened.

Every time the Worklet Service selects a worklet to execute for a specification instance, a log file is created that contains certain descriptive data about the worklet selection process. These files are stored in the *selected* folder of the worklet repository. The data stored in these files are again in XML format, and the files are named according to the following format:

CaseID_SpecificationID_RuleType_WorkItemID.xws

For example: *12_CasualtyTreatment_Selection_Treat.xws* (*xws* for Xml Worklet Selection). The identifiers in each part of the filename refer to the parent specification instance, not the worklet case instance. Also, the *WorkItemID* identifier will not appear for case-level rule types.

So, to add a new rule after an inappropriate worklet choice, the particular *selected* log file for the case that was the catalyst for the rule addition must be located and loaded into the Rules Editor.

8

From the *Add Rule* screen, click the *Open...* button to load the selection information from the relevant *selected* log file. The File Open dialog that displays will open in the *selected* folder of the repository. Select the appropriate file for the case in question then click *OK*. Note that the *selected* file chosen must be for an instance of the specification that matches the specification rule set loaded on the main screen (in other words, you can't attempt to add a new rule to a rule set that has no relation to the *xws* file opened here). If the specifications don't match, an error message will display.

Figure 17 shows the *Add Rule* form with the *selected* file *12_CasualtyTreatment_Selection_Treat.xws* loaded. The *Cornerstone Case* panel shows the case data that existed for the creation of the original rule that resulted in the selection. The *Current Case* panel shows the case data for the current case – that is, the case that is the catalyst for the addition of the new rule.

The *New Rule Node* panel is where the details of the new rule may be added. Notice that the id's of the parent node and the new node are shown as read only – the Rules Editor takes care of where in the rule tree the new rule node is to be placed, and whether it is to be added as a true child or false child node.

IMPORTANT: Since we have the case data for the original rule, and the case data for the new rule, to define a condition for the new rule it is only necessary to determine what it is about the current case that makes it require the new rule to be added. That is, it is only where the case data items *differ* that distinguish one case from the other, and further, only a subset of that differing data is relevant to the reason why the original selection was inappropriate.

For example, there are many data items that differ between the two case data sets shown in Figure 17, such as *PatientID*, *Name*, *Sex*, *Blood Pressure* readings, *Height*, *Weight* and *Age*. However, the only differing data item of relevance here is *HeartRate* – that is the only data item that, in this case, makes the selection of the *TreatRash* worklet inappropriate.

Figure 17: Add New Rule Form

8

Clicking on the line “HeartRate = 190” in the *Current Case* panel copies that line to the *Condition* input in the *New Rule Node* panel. Thus, a condition for the new rule has been easily created, based on the differing data attribute and value that has caused the original worklet selection to be invalid for this case.

Note that it is not necessary to define the rule as “Rash = True & HeartRate = 190”, as might first be expected, since this new rule will be added to the true branch of the *TreatRash* node. By doing so, it will only be evaluated if the condition of its parent, “Rash = True”, first evaluates to True. Therefore, any rule nodes added to the true branch of a parent become **exception** rules of the parent. In other words, this particular tree traversal can be interpreted as: “if Rash is True then return TreatRash **except** if HeartRate is also 190 then return ???” (where ??? = whatever worklet we decide to return for this rule – see more below).

Now, the new rule is fine if, in future cases, a patient’s heart rate will be exactly 190, but what if it is 191, or 189, or 250? Clearly, the rule needs to be amended to capture all cases where the heart rate exceeds a certain limit; say 175. While selecting data items from the *Current Case* panel is fast and easy, it is often the case that the condition needs to be further modified to correctly define the relevant rule. The *Condition* input allows direct editing of the condition.

Conditions are expressed as strings of operands and operators of any complexity, and sub-expressions may be parenthesised. The following operators are supported:

Precedence	Operators	Type
1	* /	Arithmetic
2	+ -	
3	= > < != >= <=	Comparison
4	&	Logical AND
		Logical OR
	!	Logical NOT

All conditions must finally evaluate to a Boolean value (i.e. true or false).

8

To make the condition for the new rule more appropriate, the condition “HeartRate = 190” should be edited to read “HeartRate > 175”.

After defining a condition for the new rule, the name of the worklet to be executed when this condition evaluates to true must be entered in the read-only *Conclusion* field of the *New Rule Node* panel (refer Figure 17). To select or create an appropriate worklet, click the *New...* button.

What happens next depends on whether the rule type for the tree you are adding the new rule to is of Selection type, or one of the exception types. Adding a conclusion for a Selection rule is explained below. Refer to the *Creating a New Rule Set and/or Tree Set* section below for details on adding a conclusion for the exception types.

Adding a Conclusion – Selection Rule Type

For a Selection rule tree, when the *New...* button is clicked, a dialog is displayed that comprises a drop-down list containing the names of all the worklets in the *worklets* folder of the worklet repository (refer Figure 18). An appropriate worklet for this rule may be chosen from the list, or, if none of the existing worklets are suitable, a new worklet specification may be created.

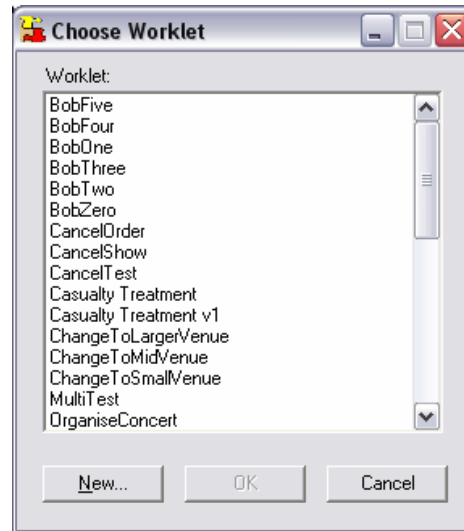


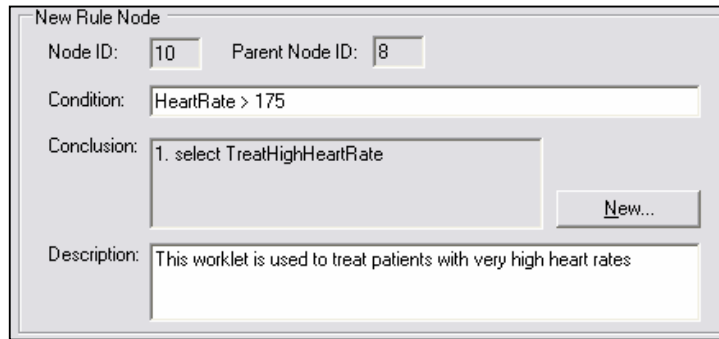
Figure 18: The Choose Worklet dialog

8

Clicking the *New...* button on this dialog will open the YAWL Editor so that a new worklet specification can be created. When defining the new worklet, bear in mind that to pass data from the original work item to the worklet, the names and data types of the variables to be passed must match those of the work item and be created as net-level variables in the worklet specification. Also, all new worklets must be saved to the *worklets* folder of the repository so that the Worklet Service can access it.

Tip: You may choose more than one worklet in this dialog simultaneously by holding down the Ctrl or Alt keys while clicking the mouse (in the usual Windows way). If you choose several worklets, when this rule is invoked for a process at runtime **all** of the worklets chosen will be launched concurrently and the process will continue only after **all** the worklets launched have completed.

When the new worklet is saved and the YAWL Editor is closed, the name of the newly created worklet will be displayed and selected in the worklet drop-down list. Click the OK button to confirm the selection and close the dialog. Figure 19 shows the *New Rule Node* panel after the definition of the example new rule has been completed. A value in the *Description* field is optional, but recommended.



The 'New Rule Node' panel is a form with the following fields:

- Node ID:** 10
- Parent Node ID:** 8
- Condition:** HeartRate > 175
- Conclusion:** 1. select TreatHighHeartRate
- Description:** This worklet is used to treat patients with very high heart rates

A 'New...' button is located to the right of the Conclusion field.

Figure 19: The New Rule Node Panel after a New Rule has been Defined

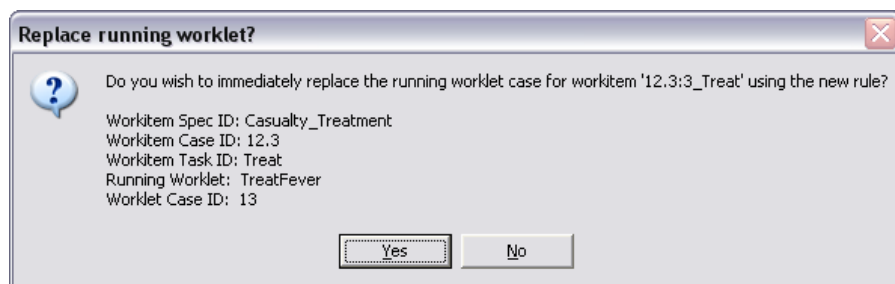
8

Once all the fields for the new rule are complete and valid, click the *Save* button to add the new rule to the rule tree.

Dynamic Replacement of an Executing Worklet

Remember that the creation of this new rule was triggered by the selection and execution of a worklet that was deemed an inappropriate choice for the current case. So, when a new rule is added, you are given the choice of replacing the executing (inappropriate) worklet instance with an instance of the worklet defined in the new rule.

After the *Save* button is clicked, a message similar to the Figure 20 is shown, providing the option to replace the executing worklet, using the new rule. The message also lists the specification and case id's of the original work item, and the name and case id of the running worklet instance.



The 'Replace running worklet?' dialog box contains the following information:

- Question:** Do you wish to immediately replace the running worklet case for workitem '12.3:3_Treat' using the new rule?
- Workitem Spec ID:** Casualty_Treatment
- Workitem Case ID:** 12.3
- Workitem Task ID:** Treat
- Running Worklet:** TreatFever
- Worklet Case ID:** 13

Buttons: Yes, No

Figure 20: Message Dialog Offering to Replace the Running Worklet

If the *Yes* button is clicked, then in addition to adding the new rule to the rule set, the Rules Editor will contact the Worklet Service and request the change. For this process to succeed, the following must apply:

- § Tomcat is currently running and the Worklet Service is correctly installed;

- § The Service URI specified in the Rules Editor configuration dialog is valid; and
- § The worklet originally chosen is currently running.

A message dialog will be shown soon after with the results of the replacement process sent from the Worklet Service back to the Rules Editor, similar to Figure 21.

If the *No* button is clicked, then the new rule is simply added to the rule set.

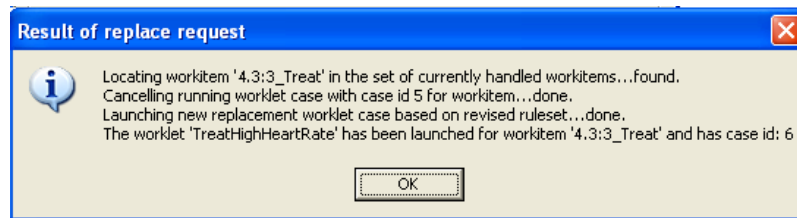


Figure 21: Result of Replace Request Dialog

Figure 22 shows the main Rules Editor screen with the new rule added in the correct place in the tree, with the current case data becoming the Cornerstone Case for the new rule.

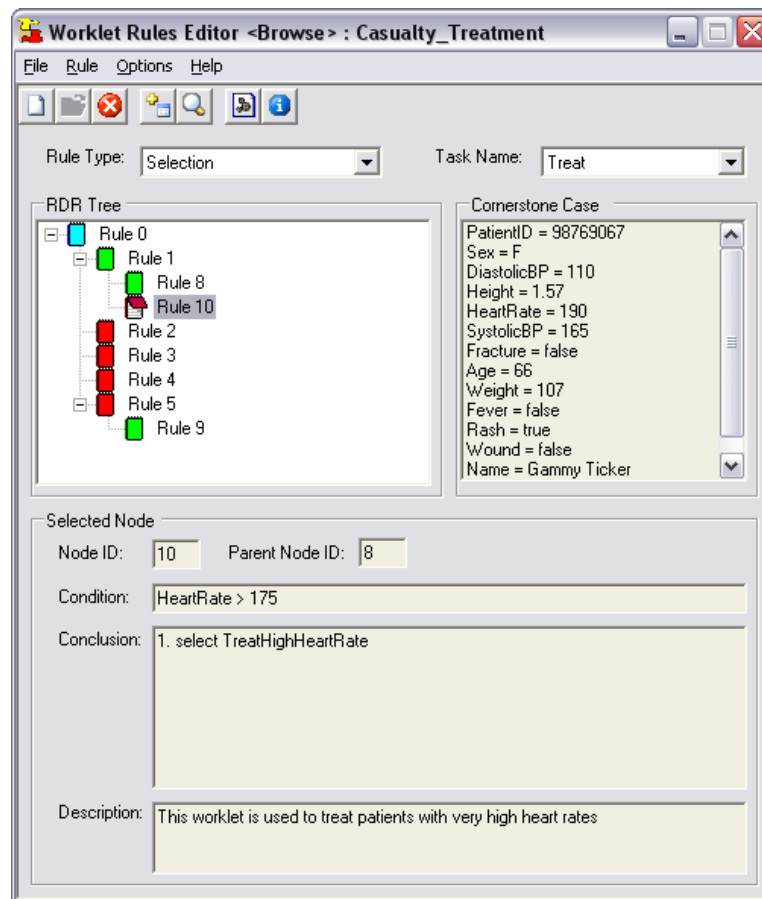


Figure 22: Main Screen after Addition of New Rule

Creating a New Rule Set and/or Tree Set

As mentioned previously, it is not necessary to create tree sets for all of the rule types, nor a rule tree for an item-level rule type encompassing every task in a specification. So, most typically, rule sets will have rules defined for a few rule types, with some tasks left undefined (remember that any events that don't have associated rules for that type of event are simply ignored).

It follows that there will be occasions where you will want to add a new tree set to a rule set for a previously undefined rule type, or add a new tree for a previously undefined task to an existing tree set. Also, when a new specification has been created, a corresponding base rule set will also need to be created (if you want to handle selections and exceptions for the new specification).

For each of these situations, the Rules Editor provides a *New Rule* form, which allows the definition of new rule trees (with any number of rule nodes) for existing tree sets (where there is a task of the specification that has not yet had a tree defined for it within the tree set); the definition of new tree sets for specifications that have not yet had a tree set defined for a particular rule type; and entirely new rule sets for specifications that have not yet had a rule set created for them.

The use of the *New Rule* form varies slightly depending on whether it is working with a new rule set or an existing rule set. This section will describe the features of the *New Rule* form for adding a new rule set, and describe how the process differs for existing rule sets as necessary.



To create a new rule set, click the *File* menu then select *New...*, or click the *New Rule* toolbar button. To add a new rule set, make sure there is no rule set file currently open in the Editor. If you are creating a new rule set, a dialog will display asking for the path and file name of the specification for which the rule set is being created (Figure 23).

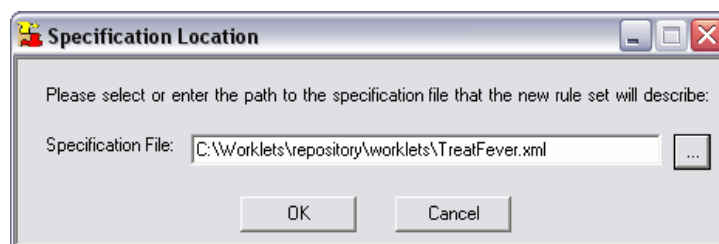


Figure 23: The Specification Location Dialog

Figure 24 shows the *Create New Rule Set* form. The form allows you create a rule set, one rule tree at a time (for the selected specification, rule type and task name). On this form:

- § The *Process Identifiers* panel is where the names of the specification, rule type and task name for the new tree are defined. The *Specification Name* input is read-only – for new rule sets it is the specification chosen via the

Specification Location dialog (Figure 23); for existing rule sets it is the specification for the rule set currently loaded into the Rules Editor. The *Rule Type* drop-down list contains all of the available rule types (i.e. all the rule types for which no or incomplete tree sets exist). For new rule sets, all rule types are available. The *Task Name* drop-down list contains all the available tasks for the selected rule type (i.e. tasks for which no tree exists in the tree set for this rule type). The *Task Name* list is disabled for case-level rule types.

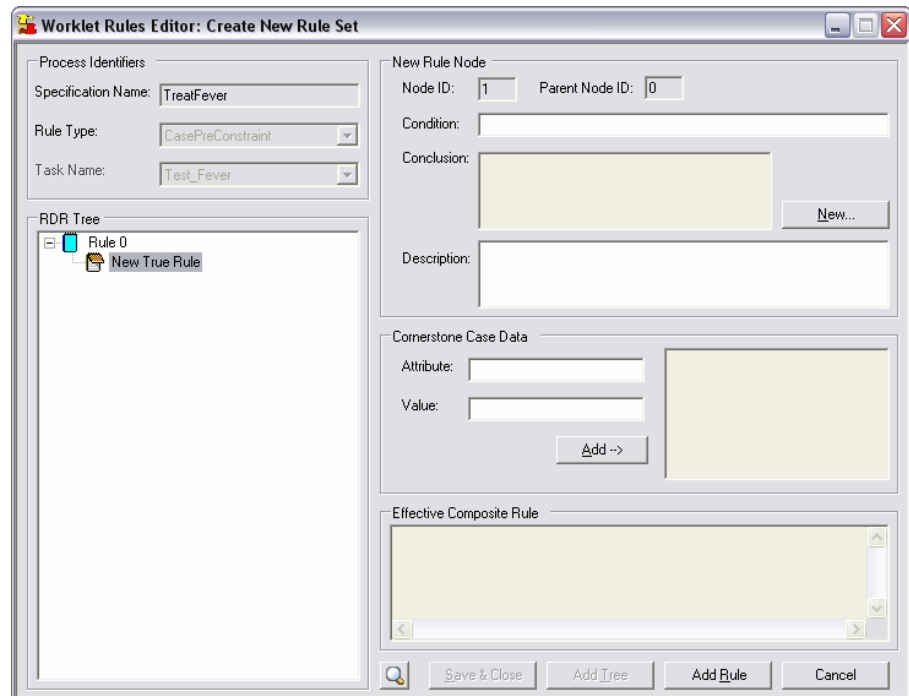


Figure 24: The Create New Rule Set Screen

- § The *New Rule Node* panel is identical to the panel on the *Add New Rule* form. Here a condition and optional description can be entered, and the conclusion for the new rule created or selected from the list (depending on the rule type – see below).
- § The *Cornerstone Case Data* panel allows a set of cornerstone data for the new rule to be defined. Add a variable name to the *Attribute* input, and give it a value in the *Value* input, then click the *Add* button to add it to the set of Cornerstone Case data. Usual naming rules apply to the data attributes: the attribute name must begin with an alpha character or underscore and contain no spaces or special characters.
- § The *Effective Composite Rule* panel displays a properly indented text equivalent of the composite condition comprising the conditions of the selected node and all ancestor nodes back to the root node – in other words, the entire composite condition that must evaluate to true for the conclusion of the selected node to be returned.

§ The *RDR Tree* panel dynamically displays graphically the new rule tree as it is being created.

New rule nodes can be added wherever there is a node on the tree that does not have a child node on both its true and false branches (except the root node which can have a true branch only). To identify possible locations to add a rule node, special ‘potential’ nodes can be seen in the *RDR Tree* panel, called “New True Node” or “New False Node”. These potential nodes are coloured yellow for easy identification.

8

To add a new rule, select the yellow new rule node where you would like the rule to be added. When you select a new rule node, the various inputs for the new rule become enabled. Refer to the *Adding a New Rule* section above for details of the types of operands and operators you can add as the condition of the new rule.

To add a conclusion to the new rule, click the *New...* button. If the currently selected rule type is *Selection*, a worklet can be added as a conclusion in the way described in the *Adding a New Rule* section. If it is one of the exception rule types, the *New...* button will display the graphical *Draw Conclusion* dialog, allowing you to build a sequence of tasks (or primitives) in an exception handling process (explained in detail below). When the conclusion sequence has been defined and the dialog closed, a text-based version of it will display in the *Conclusion* panel.

Once the new rule node has a valid condition and conclusion, and optionally some cornerstone data and a description, click the *Add Rule* button to add the rule to the tree. The new node will be displayed at the selected position on the tree with the relevant coloured node icon indicating whether it is a true or false node of its parent. New potential node add-points will also be displayed. See Figure 25 for an example of a newly created tree that has several nodes added.

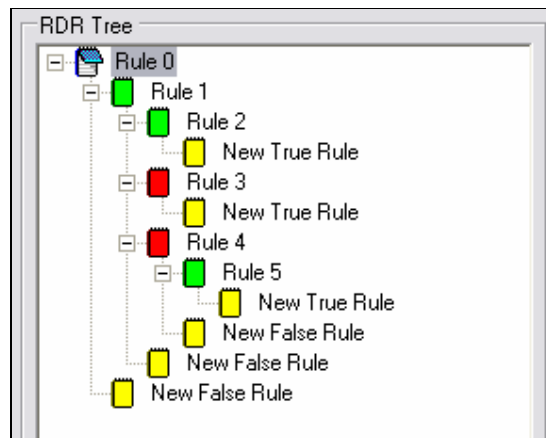


Figure 25: Creating a New Rule Tree

Repeat the add rule procedure for however many rule nodes you wish to add by clicking on the appropriate yellow node (note that when you click on a node other than yellow, its stored details are displayed in the various form inputs).

TIPS:

- § If the number of nodes starts to get a little confusing, you can check which node is the parent of the selected node by looking at its id in the *Selected Node* panel.
- § If you start to add details for a new node then change your mind about adding it, simply click on any other node (rather than clicking on the *Add Rule* button) – doing that will immediately discard any input data values. **Don't** click *Cancel* – that closes the entire form without saving anything (see below).

When you are done adding nodes, click the *Add Tree* button to add the tree you have just created to the tree set selected (via the selected *Rule Type* and *Task Name* lists).

IMPORTANT: Once you have added the newly created tree to the selected tree set using the *Add Tree* button, you will no longer be able to add nodes to the tree via the *New Rule Set* form. This is to protect the integrity of the rule set. Since each subsequent rule will be added because of an exceptional case or where the selected worklet does not fit the context of a case, the preferred method is to create the base rule set and then add rules as they become necessary via the *Add Rule* form as described earlier. In this way, added rules are based on real case data and so are guaranteed to be valid. In a similar vein, there is no option to modify or delete a rule node within a tree once the tree has been added to the rule set, since to allow it would destroy the integrity of the rule set, because the validity of child rule nodes depend on the conditions of their parents.

When a tree is added to the tree set:

- § If it is a case-level tree, the rule type that the tree represents will be removed from the *Rule Type* list. That is, the rule type now has a tree defined for it and so is no longer available for selection on the *New Rule* form.
- § If it is an item-level tree, the task name that the tree represents will be removed from the *Task Name* list. That is, the task now has a rule tree defined for it (for the selected rule type) and so is no longer available.
- § If it is an item-level tree, and all tasks now have trees defined for them for the selected rule type (i.e. this was the final task of the specification for which a tree has been defined), the rule type that the tree represents will be removed from the *Rule Type* list.

This approach ensures that rule trees can only be added where there are currently no trees defined for the selected specification.

Once the tree is added, the form resets to allow the addition of another new tree as required, by repeating the process above for a new rule type (or rule type/ task name for item-level trees).

8

After you have completed adding trees, click the *Save & Close* button to save all the additions to the rule set file. The Rules Editor will return to the main form

where the additional trees will immediately be able to be browsed.

IMPORTANT: No additions will be actually saved until the *Save & Close* button is clicked – this is to allow you the option to discard all additions, if you wish, by clicking the *Cancel* button. That is, cancelling returns to the main Editor form and discards **ALL** additions for the session; *Save & Close* returns to the main form and saves all additions.

Drawing a Conclusion Sequence

As mentioned in the *Adding a New Rule* section, adding a conclusion to a *Selection* rule is simply a matter of choosing a worklet from the list or creating a new worklet in the YAWL editor. However, when adding a conclusion for a rule type other than *Selection* (i.e. an exception rule type), an exception handling sequence needs to be defined that will manage the handling process invoked by the rule. The earlier section on the Exception Service detailed the various actions that make up the available set of exception handling ‘primitives’ or tasks that may be sequenced to form an entire handling process.

8

The *Draw Conclusion* dialog makes the process of defining an exception handling sequence easier by allowing you to create the sequence graphically. Simply select the appropriate primitive from the toolbox on the left, and then click on the drawing canvas to place the selected primitive. Figure 26 shows an example of the *Draw Conclusion* dialog.

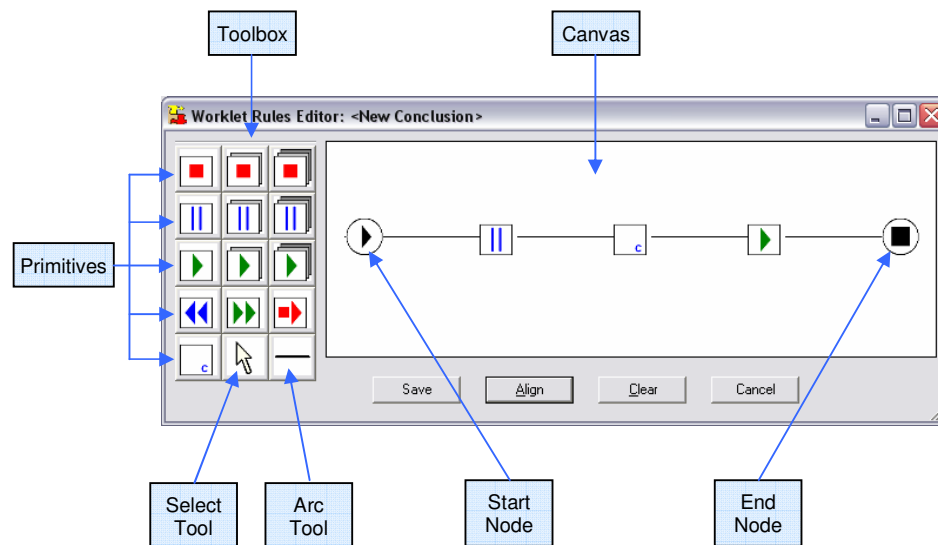


Figure 26: The Draw Conclusion Dialog

Use the *Arc Tool* to define the sequence order. First, select the *Arc Tool* in the toolbox, then click and hold on the first node, drag the mouse pointer until it is over the next node in the sequence, then release the mouse. For a conclusion to be valid (and thus allowed to be saved) there must be a direct, unbroken path from the start node to the end node (the start and end nodes are always displayed on the

canvas). Also, the conclusion will be considered invalid if there are any nodes on the canvas that are not attached to the sequence when *Save* is attempted.

Use the *Select Tool* to move placed primitives around the canvas. First, select the *Select Tool* in the toolbox, then click and drag a primitive to a new location.

The *Align* button will immediately align the nodes horizontally and equidistantly between the start and end nodes (as in figure 26).

The *Clear* button will remove all added nodes to allow a restart of the drawing process.

The *Cancel* button discards all work and returns to the previous form.

The *Save* button will save the conclusion and return to the previous form (as long as the sequence is valid).

To delete a primitive from the canvas, right click on the primitive and select *Delete* from the popup menu.

The *Compensate* primitive will, when invoked at runtime, execute a worklet as a compensation process as part of the handling process. To specify which worklet to run for this sequence, right click on the *Compensate* primitive and select *Define Worklet* from the popup menu. The *Choose Worklet* dialog will appear (identically to the *Selection* conclusion process) allowing the selection of an existing worklet or the definition of a new worklet to run as a compensatory process. Select the appropriate worklet to add it to the compensatory primitive. Note that a sequence will be considered invalid if it contains a *Compensate* primitive for which a worklet has not yet been defined.

The primitives *SuspendAllCases*, *RemoveAllCases* and *ContinueAllCases* may be limited to ancestor cases only by right-clicking on primitives of those kinds and selecting *Ancestor Cases Only* from the popup menu. Ancestor hierarchies occur where a worklet is invoked for a case, which in turn invokes a worklet, and so on. When a primitive is limited to ancestor cases, it applies the primitive's action to all cases in the hierarchy from the current case back to the original parent case, rather than all running cases of the specification.

IMPORTANT: No validation is done for the defined sequence, besides that described above. It is up to the designer of the sequence to ensure it makes sense (for example, that it doesn't try to continue a case it has previously removed).

When a valid sequence is saved, you will be returned to the previous form (i.e. either the Add Rule or New Rule form depending on where you are in the Editor). The conclusion will be displayed textually as a sequential list of tasks (Figure 27, for example).

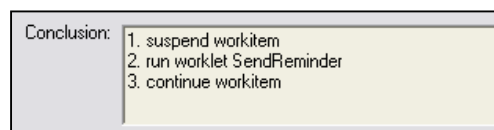


Figure 27: A Conclusion Sequence shown as Text (detail)

6. Walkthrough – Using the Worklet Service

The worklet repository that comes with the Worklet Service release contains a number of example specifications with worklet-enabled tasks, each with an associated rule set and a number of associated worklets. This section will step through the execution of several of these examples. The first two examples feature the Selection Service; the remainder the Exception Service. Knowledge of how to use the YAWL system is assumed. Before we begin, make sure the Worklet Service is correctly installed and operational, and then log into the YAWL system.

The easiest and best way to ensure the Worklet Service in ‘on-the-air’ is to go to the YAWL *Administrate* page and click on the Worklet Service’s URI link. If all’s well the service’s welcome page will be displayed (Figure 28).

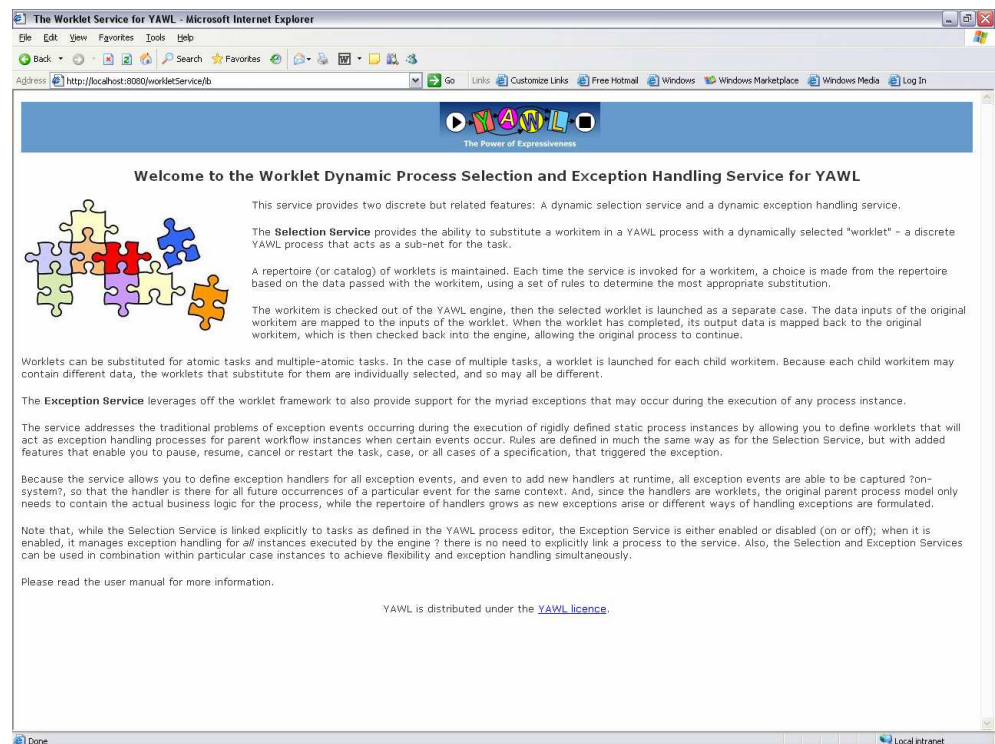


Figure 28: Welcome Page for the Worklet Service

A. Selection: Worklet-Enabled Atomic Task Example

The *Casualty Treatment* specification used in the previous sections of this manual is an example of a specification that contains an atomic task (called *Treat*) that is worklet selection-enabled. We'll run a complete instance of the example specification to see how worklet selection operates.

8

Navigate to the YAWL *Administrate* page and upload the *Casualty Treatment* specification from the *worklets* folder of the worklet repository. Then, go to the *Workflow Specifications* page and launch a *Casualty Treatment* case.

The case begins by requesting a patient id and name – just enter some data into each field then click *Submit* (Figure 29).

Figure 29: Launching a Casualty Treatment Case (detail)

8

Go to the *Available Work* page, and the first task in the case (*Admit*) will be listed as an available workitem. Make a note of the case number. Check out the *Admit* workitem.

Figure 30: Editing the Admit Workitem (detail)

The *Admit* workitem simulates an admission to the Casualty department of a hospital, where various initial checks are made of the patient. You'll see that, in

addition to the patient name and id specified earlier, there are a number of fields containing some medical data about the patient. Each field has some default data (to save time), but you may edit any fields as you wish (Figure 30). When done, click the *Submit* button.

8

Go back to the *Available Work* page and check out the next workitem, *Triage*. The *Triage* task simulates that part of the process where a medical practitioner asks a patient to nominate their symptoms. You'll see that the patient's name and id have again been displayed for identification purposes, in addition to 5 fields which approximate the medical problem. One field should be set to *True*, the others to *False*.

Let's assume the patient has a fever. Set the *Fever* field to *True*, the rest to *False*, and then submit it (Figure 31).

The screenshot shows a web form titled "Triage". It contains the following fields and controls:

- Patient ID**: A text input field containing "123456".
- Fever**: A radio button group with "true" selected (indicated by a green dot) and "false" unselected.
- Rash**: A radio button group with "true" unselected and "false" selected (indicated by a green dot).
- Wound**: A radio button group with "true" unselected and "false" selected (indicated by a green dot).
- Name**: A text input field containing "Iva Payne".
- Abdominal Pain**: A radio button group with "true" unselected and "false" selected (indicated by a green dot).
- Fracture**: A radio button group with "true" unselected and "false" selected (indicated by a green dot).
- Submit**: A button at the bottom of the form.

Red asterisks (*) are placed to the right of the Patient ID, Name, and each of the five symptom radio button groups, indicating required fields.

Figure 31: Editing the Triage Workitem (detail)

There is nothing special about the first two tasks in the process; they are standard YAWL tasks and operate as expected. However, the next task, *Treat*, has been associated (using the YAWL Editor) with the Worklet Selection Service. The *Treat* task simulates that part of the process that follows the collection of patient data and actually treats the patient's problem.

Of course, there are many medical problems a patient may present with, and so there are just as many treatments, and some treatment methods are vastly different to others. In a typical workflow process, this is the part of the process where things could get very complicated, particularly if we tried to build every possible treatment as a conditional branch into the process model.

The Worklet Service greatly simplifies this problem, by providing an extensible repertoire of discrete workflow processes (worklets) which, in this example, each handle the treatment of a particular medical problem. By examining the case data collected in the earlier tasks, the Worklet Service can launch, as a separate case, the particular treatment process for each case.

This method allows for a simple expression of the task in the ‘parent’ process (i.e. a single atomic *Treat* task signifies the treatment of a patient, whatever the eventual treatment process may be) as well as the ability to add to the repertoire of worklets at any time as new treatments become available, without having to modify the original process.

When the *Triage* workitem is submitted, the next task in the process, *Treat*, becomes enabled. Because it is worklet-enabled, the Worklet Selection Service is notified. The Selection Service checks to see if there is a set of rules associated with this workitem, and if so the service checks out the workitem.

When this occurs, the YAWL Engine marks the workitem as executing (externally to the Engine) and waits for the workitem to be checked back in. In the meantime, the Worklet Service uploads the relevant specification for the worklet chosen as a substitute for the workitem and launches a new case for the specification. When the worklet case completes, the Worklet Service is notified of the case’s completion, and the service then checks the original workitem back into the Engine, allowing the original process to continue.

8

We have completed editing the *Triage* workitem and clicked the *Submit* button. Go to the *Available Work* page. Instead of seeing the next workitem listed (i.e. *Treat*), we see that *Test Fever*, the first workitem in the *TreatFever* process, is listed in its place (Figure 32). The *TreatFever* process has been chosen by the Worklet Service to replace the *Treat* workitem based on the data passed to the service.

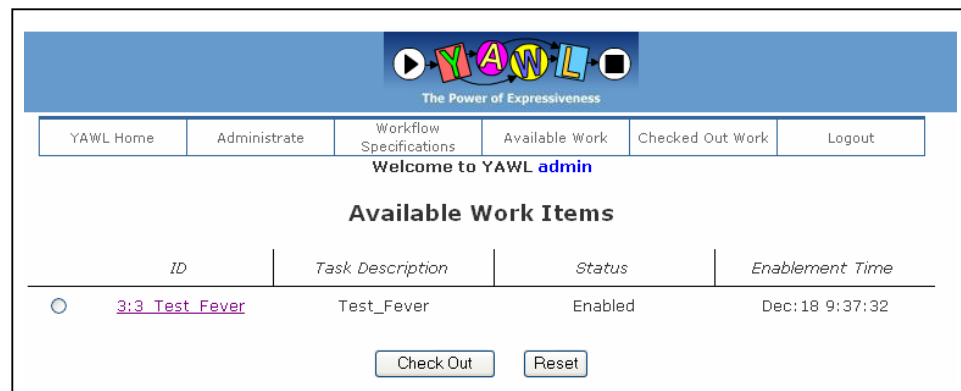


Figure 32: New Case Launched by the Worklet Service

8

Note that the case id for the *Test Fever* workitem is different to the case id of the parent process. Worklets run as completely different cases to the parent process, but the Worklet Service keeps track of which worklets are running for which parent cases. Go to the *Workflow Specifications* page to see that a *Casualty Treatment* case is still running, and that the *TreatFever* specification has been loaded and it also has a case running (Figure 33).

8

Go back to the *Available Work* page and check out the *Test Fever* workitem. The *Test Fever* workitem has mapped the patient name and id values, and the particular symptom – fever – from the *Treat* workitem checked out by the

Worklet Service. In addition, it has a *Notes* field where a medical practitioner can enter observations about the patient's condition (Figure 34). Enter some information into the *Notes* field, and then submit it.

The screenshot shows the YAWL admin interface. At the top, there is a navigation bar with links: YAWL Home, Administrate, Workflow Specifications, Available Work, Checked Out Work, and Logout. Below the navigation bar, it says "Welcome to YAWL admin". The main section is titled "Active YAWL Specifications". It contains a table with the following columns: Specification ID, Spec Name, Documentation, XML, and Cases.

Specification ID	Spec Name	Documentation	XML	Cases
TreatFever	Treat Fever	Worklet to treat a fever	View TreatFever	3
Casualty Treatment	Casualty Treatment	A simple medical treatment process designed to test and demonstrate the Worklet Dynamic Process Selection Service within the YAWL engine.	View Casualty Treatment	2

At the bottom of the table, there are two buttons: "Launch Case" and "Reset".

Figure 33: *TreatFever Specification Uploaded and Launched*

8

Check out the next workitem, *Treat Fever*, and then edit it. This workitem has two additional fields, Treatment and Pharmacy, where details about how to treat the condition can be entered (Figure 35). Enter some data here, and then submit it.

The screenshot shows the "Test Fever" workitem form. It has the following fields:

- Patient ID: 123456
- Notes: bitten by spider
- Fever: ☒ true ☐ false
- Name: Iva Payne

Each field has a red asterisk indicating it is required. At the bottom, there is a "Submit" button.

Figure 34: *Test Fever Workitem (detail)*

Figure 35: *Treat Fever Workitem (detail)*

When the *Treat Fever* workitem is submitted, the worklet case is completed. The Worklet Service maps the output data from the worklet case to the matching variables of the original *Treat* workitem, then checks that workitem back in, allowing the next workitem in the *Casualty Treatment* process, *Discharge*, to execute.

8

Go to the *Available Work* page, and you'll see that the *Discharge* workitem is available (Figure 36). Edit it to see that the data collected by the *TreatFever* worklet has been mapped back to this workitem. Submit it to complete the case.

Figure 36: *Discharge Workitem with Data Mapped from TreatFever Worklet*

B. Selection: Worklet-Enabled Multiple Instance Atomic Task Example

This walkthrough takes the *List Maker* example from the YAWL Editor User Manual and worklet-enables the *Verify List* task to show how multiple instance atomic tasks are handled by the Worklet Selection Service.

The specification is called *wListMaker*. The only change made to the original *List Maker* specification was to associate the *Verify List* task with the Worklet Service using the YAWL Editor. Figure 37 shows the specification.

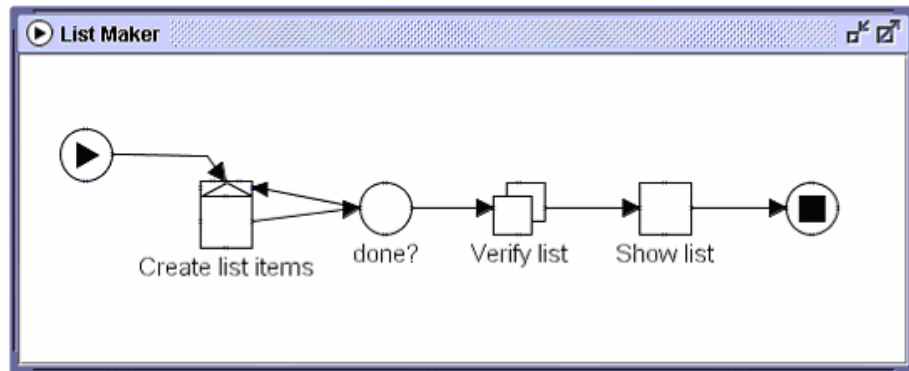


Figure 37: The *wListMaker* Specification

8

Go to the *Administrate* page and upload the *wListMaker* specification from the *worklets* folder of the worklet repository. Then, go to the *Workflow Specifications* page and launch an instance of *wListMaker*.

When the case begins, enter 3 values for the *Bob* variable, as shown in Figure 38 – you will have to click the *Insert after selected* button twice to get three input fields. Make sure you enter the values “one”, “two” and “three” (without the quotes and in any order). Submit the form.

Figure 38: Start of *wListMaker* Case with Three ‘Bob’ Values Entered (detail)

8

Check out and edit the *Create List Items* workitem. Since the values have already been entered there is no more to do here, so click the *Submit* button to continue.

The next task is *Verify List*, which has been associated with the Worklet Selection Service. Since this task is a multiple-instance atomic task, three child workitem instances of the task are created, one for each of the *Bob* values entered previously. The Worklet Service will determine that it is a multiple instance atomic task and will treat each child workitem instance separately, and will launch the appropriate worklet for each based on the data contained in each. Since the data in each child instance is different in this example, the Worklet Service starts 3 different worklets, called *BobOne*, *BobTwo* and *BobThree*. Each of these worklets contains only one task.

8

Go to the *Available Work* page. There are three workitems listed, each one the first workitem of a separate case (see Figure 39).

YAWL Home	Administrate	Workflow Specifications	Available Work	Checked Out Work	Logout
Welcome to YAWL admin					
Available Work Items					
ID	Task Description	Status	Enablement Time		
5:2 Get Bob One	Get_Bob_One	Enabled	Dec:18 12:36:57		
7:2 Get Bob Two	Get_Bob_Two	Enabled	Dec:18 12:36:59		
6:2 Get Bob Three	Get_Bob_Three	Enabled	Dec:18 12:36:58		
		<input type="button" value="Check Out"/>	<input type="button" value="Reset"/>		

Figure 39: Workitems from each of the 3 Launched Worklet Cases

8

Go to the *Workflow Specifications* page to see that the *BobOne*, *BobTwo* and *BobThree* specifications have been uploaded and launched by the Worklet Service as separate cases (Figure 40 – note the case numbers).

Go back to the *Available Work* page and check out all three workitems. Edit each of the *Get_Bob* workitems, and modify the values as you wish – for this walkthrough, we'll change the values to “one – five”, “two – six” and “three – seven” respectively.

As you edit and submit each *Get_Bob* workitem, notice that the corresponding *Verify List* workitem is automatically checked in by the Worklet Service (you'll only see this if you are logged in as ‘admin’). Since the *Bob* worklets contains only one task, editing and submitting this workitem also completes the worklet case.

YAWL Home	Administrative	Workflow Specifications	Available Work	Checked Out Work	Logout
Welcome to YAWL admin					
Active YAWL Specifications					
Specification ID	Spec Name	Documentation	XML	Cases	
<input type="radio"/> BobThree	Bob Three	Worklet to enact when bob is three	View BobThree	6,	
<input type="radio"/> BobTwo	Bob Two	Worklet to enact when bob is two	View BobTwo	7,	
<input type="radio"/> BobOne	Bob One	Worklet to enact when bob is one	View BobOne	5,	
<input type="radio"/> TreatFever	Treat Fever	Worklet to treat a fever	View TreatFever		
<input type="radio"/> wListMaker	List Maker Example (worklet enabled)	A process to demonstrate how worklets handle a multiple task	View wListMaker	4,	
<input type="radio"/> Casualty_Treatment	Casualty Treatment	A simple medical treatment process designed to test and demonstrate the Worklet Dynamic Process Selection Service within the YAWL engine.	View Casualty_Treatment		
<input type="button" value="Launch Case"/> <input type="button" value="Reset"/>					

Figure 40: 'Bob' Specifications Loaded and Launched by the Worklet Service

8

After the third workitem has been edited and submitted, and so the third *Verify List* workitem is checked back into the Engine by the Worklet Service, the Engine determines that the *Verify List* workitem has completed and so the original process continues to its final workitem, *Show List*.

Check out and edit the *Show List* workitem to show the changes made in each of the *Get_Bob* worklets have been mapped back to the original case (Figure 41).

Show List

User List

Bob

☒ Bob *

☐ Bob *

☐ Bob *

Figure 41: The Show List Workitem Showing the Changes to the Data Values

C. Exception: Constraints Example

This walkthrough uses a specification called *OrganiseConcert* to demonstrate a few features of the Worklet Exception Service. The *OrganiseConcert* specification is a very simple process modelling the planning and execution of a rock concert. Figure 42 shows the specification as it appears in the YAWL Editor.

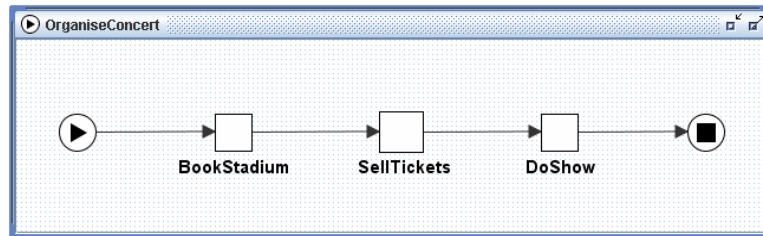


Figure 42: The *OrganiseConcert* Specification

8

First, ensure the Exception Service is enabled (see Section 2 for details). Navigate to the YAWL *Administrate* page and upload the *OrganiseConcert* specification from the *worklets* folder of the worklet repository. Then, go to the *Workflow Specifications* page and launch an *OrganiseConcert* case.

As soon as the Engine launches the case, it notifies the Exception Service via a *PreCaseConstraint* event. If the rule set for *OrganiseConcert* contains a rule tree for pre-case constraints, that tree will be queried using the initial case data to determine whether there are any pre-constraints not met by the case. In this example, there are no pre-case constraint rules defined, so the notification is simply ignored.

Tip: To follow what is happening, watch the log output in the Tomcat command window – both the exception and selection services log all interactions between themselves and the Engine to the Tomcat window and to a log file (found in the *logs* folder of your Tomcat installation). See *Appendix B* of this manual for a sample log output for this walkthrough.

Pre-case constraints can be used, amongst other things, to ensure case data is valid or within certain ranges before the case proceeds; can be used to run compensatory worklets to correct any invalid data; or may even be used to cancel the case as soon as it starts (in certain circumstances). As a trivial example of the last point, launch an instance of the *Casualty Treatment* specification discussed in *Walkthrough A*, and enter “smith” for the patient name when the case starts. The *Casualty Treatment* rule set contains a pre-case constraint rule to cancel the case if the patient’s name is “smith” (presumably smith is a hypochondriac!) This is also an example of exception rules and selection rules being defined within the same rule set.

Directly following the pre-case event, the Engine notifies the Service of a *PreItemConstraint* for the first workitem in the case (in this case, *Book Stadium*). The pre-item constraint event occurs immediately the workitem becomes enabled

(i.e. ready to be checked out or executed). Like pre-case constraint rules, pre-item rules can be used to ensure workitems have valid data before they are executed. The entire set of case data is made available to the Exception Service – thus the values of any case variables may be queried in the ripple-down rules for any exception type rule. While there are pre-item constraint rule trees defined in the rule set, there are none for the *Book Stadium* task, so this event is also ignored by the service.

8

The *Book Stadium* workitem may be checked out in the normal fashion. This workitem captures the seating capacity, cost and location of the proposed rock concert. These may be changed to any valid values, but for the purposes of this example, just accept the default values as given (Figure 43).

Book Stadium	
Venue Cost	100000.00 *
Seating	25000 *
Venue Name	ANZ Stadium *
<input type="button" value="Submit"/>	

Figure 43: The *Book Stadium* Workitem (detail)

When the workitem is submitted, a *PostItemConstraint* event is generated by the Engine. There are no post-item constraint rules for this workitem, so again the event is just ignored. Then, a pre-item constraint notification is received for the next workitem (*Sell Tickets*). This workitem records the number of tickets sold, and the price of each ticket. Enter a price of \$100 per ticket, and 12600 as the number of tickets sold, and then submit the workitem (Figure 44).

Sell Tickets	
Ticket Cost	100 *
Tickets Sold	12600 *
<input type="button" value="Submit"/>	

Figure 44: The *Sell Tickets* Workitem (detail)

Notice that the entered number of tickets sold (12600) is slightly more than 50% of the venue's seating capacity (25000). The next workitem, *Do Show*, does have a pre-item constraint rule tree, and so when it becomes enabled, the rule tree is queried. The effective composite rule for *Do Show*'s pre-item tree (as viewed in the Rules Editor), is:

Effective Composite Rule
if TicketsSold < (Seating * 0.75) then suspend workitem; run worklet ChangeToMidVenue; continue workitem except if TicketsSold < (Seating * 0.5) then suspend workitem; run worklet ChangeToSmallVenue; continue workitem except if TicketsSold < (Seating * 0.2) then suspend case; run worklet CancelShow; remove case

Figure 45: Effective Composite Rule for Do Show's Pre-Item Constraint Tree

In other words, when *Do Show* is enabled and the value of the case data attribute "TicketsSold" is less than 75% of the seating capacity of the venue, we would like to suspend the workitem, run the compensatory worklet *ChangeToMidVenue*, and then, once the worklet has completed, continue (or unsuspend) the workitem. Following the logic of the ripple-down rule, if the tickets sold are also less than 50% of the capacity, then we want instead to suspend the workitem, run the *ChangeToSmallVenue* worklet, and then unsuspend the workitem. Finally, if there has been less than 20% of the tickets sold, we want instead to suspend the entire case, run a worklet to cancel the show, and then remove (i.e. cancel) the case.

In this example, the first rule's condition evaluates to true, for a Tickets Sold value of 12600 and a seating capacity of 25000, so the child rule node on the true branch of the parent is tested. Since this node's condition evaluates to false for the case data, the rule evaluation is complete and the last true node returns its conclusion.

The result of all this can be seen in the *Available Work* screen of the worklist. The *Do Show* workitem is marked as "Suspended" and thus is unable to be selected for checking out; while the *ChangeToMidVenue* worklet has been launched and its first workitem, *Cancel Stadium*, is enabled and may be checked out. By viewing the log file, you will see that the *ChangeToMidVenue* worklet is being treated by the Exception Service as just another case, and so receives notifications from the Engine for pre-case and pre-item constraint events also.

8

Check out *Cancel Stadium*, accept the default values, and submit. Notice that the worklet has mapped the data attributes and values from the parent case. Next, check out the *Book Ent Centre* workitem – by default, it contains the data values mapped from the parent case. Since we are moving the concert to a smaller venue, change the values to match those in Figure 46, then submit the workitem.

Book Ent Centre	
Venue Cost	50000.00 *
Seating	15000 *
Venue Name	Ent Centre *
<input type="button" value="Submit"/>	

Figure 46: The Book Ent Centre Workitem (detail)

The third workitem in the worklet, *Tell Punters*, is designed for the marketing department to advise fans and existing ticket holders of the change in venue.

Check the workitem out from *Available Work*. Notice that the values here are read-only (since this item is meant to be a notification only, the person assigned does not need to change any values). This is the last workitem in the worklet, so when that is submitted, the engine completes the case and notifies the Exception Service of the completion, at which time the service completes the third and final part of the exception handling process, i.e. continuing or unsuspending the *Do Show* workitem so that the parent case can continue.

Back at the *Available Work* screen, the *Do Show* workitem is now shown as enabled and thus is able to be checked out. Check it out now and notice that the data values entered in the worklet's *Book Ent Centre* workitem have been mapped back to the parent case.

D. Exception: External Trigger Example

It has been stated that every case instance involves some deviation from the standard process model. Sometimes, events occur completely removed from the actual process model itself, but affect the way the process instance proceeds. Typically, these kinds of events are handled “off-system” so there is no record of them, or the way they were handled, kept for future executions of the process specification.

The Worklet Exception Service allows for such events to be handled on-system by providing a means for exceptions to be raised by users externally to the process itself. The *Organise Concert* specification will again be used to illustrate how external triggers work.

8

Go to the *Administrate Screen* and launch another instance of the *Organise Concert* specification. Execute and submit the first workitem.

To raise a case-level external exception, go to the *Workflow Specifications* screen, and click on the current case id for the *Organise Concert* case (Figure 47).

Active YAWL Specifications				
Specification ID	Spec Name	Documentation	XML	Cases
TreatFever	Treat Fever	Worklet to treat a fever	View_TreatFever	
ChangeToMidVenue	ChangeToMidVenue	Action taken if ticket sales less than expected	View_ChangeToMidVenue	
Casualty_Treatment	Casualty Treatment	A simple medical treatment process designed to test and demonstrate the Worklet Dynamic Process Selection Service within the YAWL engine.	View_Casualty_Treatment	
OrganiseConcert	Organise Concert	Example used to test workletService Exception Handling	View_OrganiseConcert	22
		Launch Case	Reset	

Figure 47: Workflow Specifications Screen, *OrganiseConcert* case running

The *Case Viewer* Screen appears with 4 buttons available (if it only has two buttons, the exception service has not been correctly enabled for the worklist; review the installation section at the beginning of this document for details on

how to enable the service in the worklist). Click on the *Raise Exception* button (Figure 48).

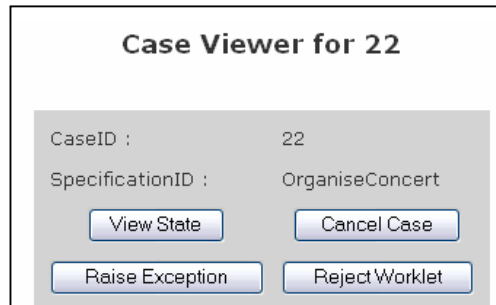


Figure 48: Case Viewer Screen

The *Raise Case Level Exception Screen* is now displayed. This screen is a member of a set of Worklet Service add-in screens for the worklist (note the worklet logo in the top left of the banner, identifying it as a worklet add-in screen). Before this screen is displayed, the Exception Service retrieves from the rule set for the selected case the list of existing external exception triggers (if any) for the case's specification. See Figure 49 for the list of case-level external triggers defined for the *Organise Concert* specification.

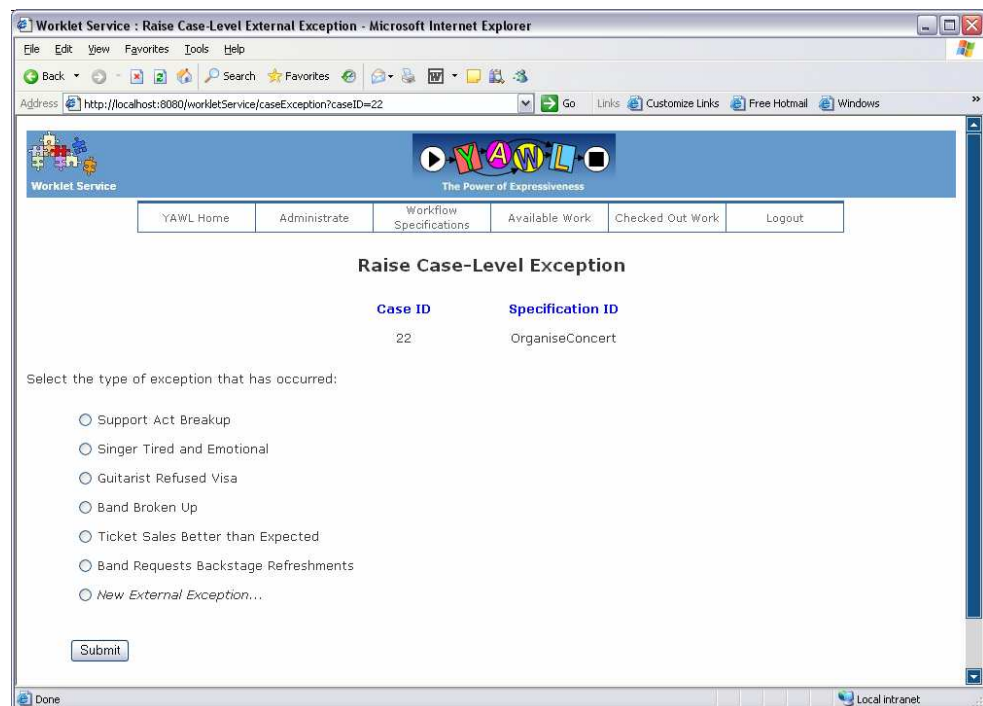


Figure 49: Raise Case-Level Exception Screen (*Organise Concert* example)

This list contains all of the external triggers either conceived when the specification was first designed or added later as new kinds of exceptional events occurred and were added to the rule set. Notice that at the bottom of the list, the option to add a New External Exception is provided – that option is explained in detail in *Walkthrough F*.

For this example, let's assume the band has requested some refreshments for backstage. Select that exception trigger and submit the form. When that exception is selected, the conclusion for that trigger's rule is invoked by the service as an exception handling process for the current case. You are immediately taken to the *Available Work* form where it can be seen that the parent case has been suspended and the first workitem of the compensatory worklet, *Organise Refreshments*, has been enabled (Figure 50).

Available Work Items			
ID	Task Description	Status	Enablement Time
<input checked="" type="radio"/> 23:Buy_M_and_Ms_5	Buy_M_and_Ms	Enabled	Sep:12 14:00:54
<input type="radio"/> 22:SellTickets_3	SellTickets	Suspended	Sep:12 13:40:04
<input type="button" value="Check Out"/> <input type="button" value="Reset"/> <input type="button" value="Raise Exception"/>			

Figure 50: Available Work Items after External Exception Raised

Organise Refreshments informs the staff member responsible to buy a certain number of bags of M & M's (first workitem), then to remove the candies of a specified colour, before delivering them to the venue (mapped in from the parent case). Once the worklet has completed, the parent case is continued.

Item-level external exceptions can be raised from the *Available Work* screen by selecting the relevant workitem via the radio button to its left, then clicking the *Raise Exception* button. You will be taken to the *Raise Item Level Exception* screen where the procedure is identical to that described for case-level exceptions, except that the item-level external exception triggers, if any, will be displayed.

External exceptions can be raised at any time during the execution of a case – the way they are handled may depend on how far the process has progressed (via the defining of appropriate rule tree or trees).

E. Exception: Timeout Example

YAWL provides a Time Service which, when a workitem is associated with it, will check out the workitem, wait until a pre-defined time span has elapsed or a certain date/time reached, then check the item back in. Effectively, the time Service allows a workitem to act as a timer on a process branch; typically one or more workitems execute in parallel to the time service workitem. If the deadline is reached, a timeout occurs for that item.

When a workitem times out, the Engine notifies the Exception Service and informs it of all the workitems running in parallel with the timed out item. Thus, rule trees can be defined to handle timeout events for all affected workitems (including the timed out item itself).

The specification *Timeout Test 3* gives an example of how a timeout exception may be handled (Figure 51). Upload the specification via the *Administrate* screen, and then launch the case.

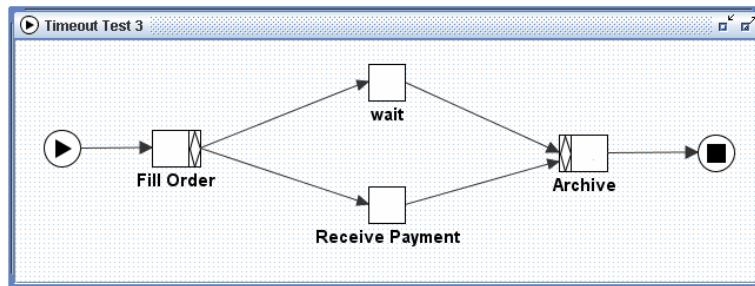


Figure 51: The Timeout Test 3 Specification

The first workitem, *Fill Order*, simulates a basic purchase order for a bike. Check out the *Fill Order* workitem, accept the default values, and submit it. Once the order has been filled, the process waits for payment to be received for the order, before it is archived. The wait task is associated with the Time Service, and so merely waits for some time span to pass. For the purposes of this example, the wait time is set to 5 seconds.

While the deadline is reached, the Engine notifies the Exception Service of the timeout event. The timeout tree set is queried for both the *wait* task and the parallel *Receive Payment* workitem. There is a tree defined for the *Receive Payment* task with a single rule (see Figure 52).

Notice the rule's condition: *isNotCompleted(this)*:

- § *isNotCompleted* is an example of a defined function that may be used as (or as part of) conditional expressions in rule nodes.
- § *this* is a special variable created by the Worklet Service that refers to the workitem that the rule is defined for and contains, amongst other things, all of the workitem's data attributes and values.

Tip: The Worklet Service provides developers with an easily extendible class where functions can be defined and then used in conditions. See *Appendix A* for more information about defining functions.

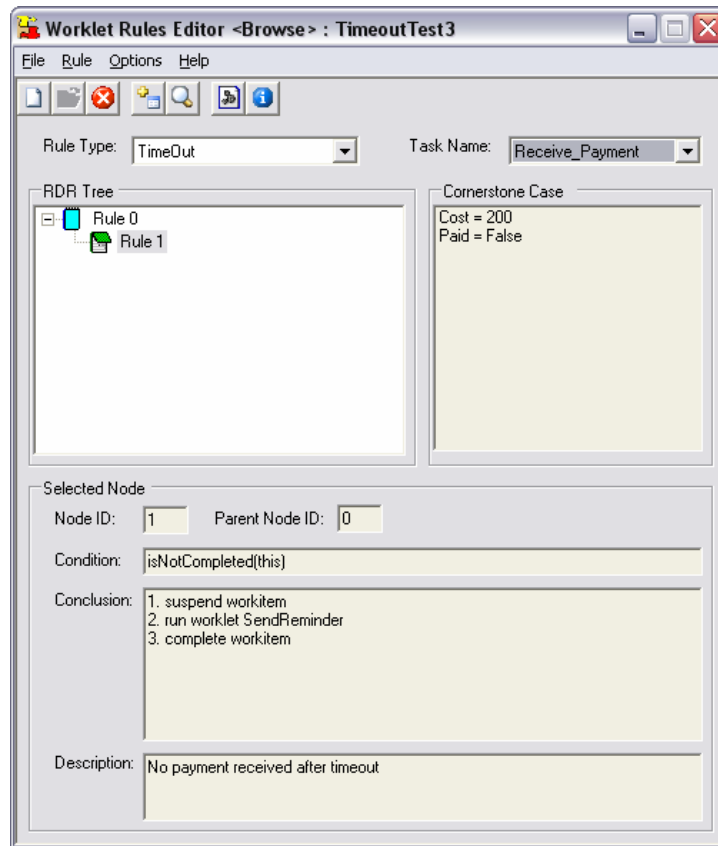


Figure 52: Rules Editor Showing Single Timeout Rule for Receive_Payment Task

In this case, the condition tests if the workitem *Receive_Payment* has not yet completed (i.e. if it has a status of *Fired*, *Enabled*, *Executing* or *Suspended*). If it hasn't completed (thus payment for the order has not yet been received) then the conclusion will be executed as an exception handling process, including the launching of the worklet *SendReminder*.

The *SendReminder* worklet consists of three tasks: *Send Request*, and the parallel tasks *wait* and *Receive Reply* – the *wait* task in this specification is again associated with the Time Service and again, for the purposes of the example, waits for 5 seconds before timing out. When the *wait* task times out, the Exception Service is notified. There is a single timeout rule for the *Receive Reply* task – its condition is again *isNotCompleted(this)* but this time, the rule's conclusion looks like this:

Condition:	isNotCompleted(this)
Conclusion:	<ol style="list-style-type: none"> 1. suspend case 2. run worklet CancelOrder 3. remove ancestorCases

Figure 53: Rule detail for Receive Reply

When this conclusion is executed as an exception handling process, the *Available Work* Screen in the YAWL worklist now looks like this:


Available Work Items			
ID	Task Description	Status	Enablement Time
30:Receive_Payment_3	Receive_Payment	Suspended	Sep:13 10:48:36
 32:File_Cancellation_3	File_Cancellation	Enabled	Sep:13 10:49:00
31:Receive_Reply_5	Receive_Reply	Suspended	Sep:13 10:48:54

Figure 54: Available Work After CancelOrder Worklet Launched

File Cancellation is the first task of the *Cancel Order* worklet. What we now have is a hierarchy of worklets: case 30 (*TimeoutTest3*) is suspended pending completion of worklet case 31 (*Send Reminder*) which itself is suspended pending completion of worklet case 32 (*Cancel Order*). Worklets can invoke child worklets to any depth. Notice the third part of the handling process: “remove ancestorCases”. Ancestor Cases are all cases from the current worklet case back up the hierarchy to the original parent case that began the exception chain (as opposed to “allCases” which refers to all currently executing cases of the same specification as the case which generates the exception). So, when the *Cancel Order* worklet completes, the *Send Reminder* case and the original parent *Timeout Test 3* are both cancelled by the Exception Service.

F: Rejecting a Worklet and/or Raising a New External Exception

The processes involved in rejecting a worklet (launched either as a result of the Selection or the Exception Service) and raising a new external exception (that is, an external exception which has not yet been defined – formally an *unexpected exception*) are virtually identical and so are discussed together in this section.

When the service launches a worklet, it selects the most appropriate one based on the current case context and the current rule set for the parent case. As discussed previously in this manual, there may be occasions where the selected worklet does not best handle the current case’s context (perhaps because of a new business rule or a more efficient method of achieving the goal of a task being found). In any event, a worker may choose to reject the worklet that was selected.

IMPORTANT: The rejection of a selected worklet is a legitimate and expected occurrence. Each rejection allows for the addition of a new exception rule (or a rule on the true branch of its parent) thus creating a ‘learning’ system where all events are handled on line. When the new rule is added as a result of the rejection, it will return the correct worklet for every subsequent case that has a similar context. Thus rejecting a worklet actually refines the rule set for a specification.

To reject a selected worklet, go to the *Workflow Specifications* screen and click on the case id of the worklet you wish to reject. At the *Case Viewer* screen, click the *Reject Worklet* button (see Figure 48). You will be redirected to the *Reject Worklet Selection* screen, another Worklet Service add-in screen (Figure 55). This

screen displays the Specification and Case ID for the selected worklet. Enter a proposed title (or name) for the new worklet and an explanation of reason for the rejection (in plain text), and then submit the form.

Figure 55: *Reject Worklet Selection Screen*

To raise an unexpected exception at the case-level, follow the same process to the *Case Viewer* screen, but click the *Raise Exception* button. On the *Raise Case-Level Exception* screen (discussed in *Walkthrough D*), select *New External Exception* from the list and submit the form. You will be redirected to the *Define New Case Level Exception* screen. Enter a proposed title, a description of the scenario (what has happened to cause the exception) and a (optionally) a proposal or description of how the new worklet will handle the exception (in plain text), and then submit the form. See Figure 56 for an example using the *Organise Concert* specification. Raising an item-level exception is identical, except that the *Raise Exception* button is clicked on the *Available Work* screen, rather than the *Case Viewer* screen.

The information entered on the form is sent to a Worklet Service Administrator, who will action the rejection or new exception by adding a new rule to the rule set and (optionally) having the Rules Editor notify the service to reselect the new worklet using the updated rule set (see the Section 5 on the Rules Editor for more detail). The process requires a user with administration authority to action the rejection request, rather than allow all users access to update rule sets.

Note: Rejecting a worklet selection or raising a new unexpected exception will automatically suspend the parent case until such time as the rejection or unexpected exception is actioned by an administrator.

Figure 56: Example of a New Case-Level Exception Definition

If you are logged into the YAWL worklist as an admin user, and the exception service is enabled, you will notice an extra link on the top menu of most of the worklist screens called *Worklet Admin Tasks* (for example Figure 56 above). This link allows administrators to view the current list of outstanding worklet rejections and requests for new exception handlers. It also allows administrators to view the details of each outstanding rejection and exception request and to mark it as completed (removing them from the list) after it has been actioned (figure 57).


Worklet Service Administration Tasks		
	Title	Case ID
	Try Another Reminder	32
		Rejected Worklet Selection
View Details		Completed

Figure 57: Administration Tasks Screen (detail)

Appendix A: Defining New Functions for Rule Node Conditions

In Section 5, we saw how rule conditions could be defined using a combination of arithmetic operators and operands consisting of data attributes and values found in workitems and at the case level of process instances. In *Walkthrough D*, an example of a defined function was given (*isNotCompleted*), using the special variable *this*.

The Worklet Service provides a discrete class that enables developers to extend the availability of such defined functions. That is, a developer may define new functions that can then be used as (or as part of composite) conditional expressions in rule nodes. That class is called *RdrConditionFunctions* – the source code for the class can be found in the *au.edu.qut.yawl.worklet.support* package. Currently, this class contains a small number of examples to give developers an indication of the kinds of things you can do with the class and how to create your own functions.

The class code is split into four sections:

- § Header;
- § Execute Method;
- § Function Definitions; and
- § Implementation.

To successfully add a function, these rules must be followed:

1. Create the function (i.e. a normal Java method definition) and add it to the 'function definitions' section of the code. Ensure the function:

- § is declared with the access modifier keywords *private static*; and
- § returns a value of *String* type.

2. Add the function's name added to the array of '_functionNames' in the header section of the code.
3. Add a mapping for the function in the 'execute' method, using the examples as a guide.

Once the function is added, it can be used in any rule's conditional expression.

Let's use the *max* function as a simple example walkthrough (to be read in conjunction with the source code for the class). The first thing to do is define the actual function in the function definition section. Here's the entire function:

```
public static String max(int x, int y) {
    if (x >= y) return String.valueOf(x) ;
    else return String.valueOf(y) ;
}
```

Notice that the function has been declared as *private static* and returns a *String* value. Next, the name of the function, *max*, has to be added as a *String* value to the *_functionNames* array in the header section of the code:

```
public static final String[] _functionNames = { "max",
                                                "min",
                                                "isNotCompleted",
                                                "today" } ;
```

Finally, we need to map the function name to the *execute* method, which acts as the interface between the class's functions and the Worklet Service. The *execute* method receives as arguments the name of the function to execute, and a *HashMap* containing the function's parameters (all are passed as *String* values). The *execute* method is essentially an *if...else if* block, the sub-blocks of which call the actual functions defined. This is the section of the *execute* method for the *max* function:

```
else if (name.equalsIgnoreCase("max")) {
    int x = getArgAsInt(args, "x");
    int y = getArgAsInt(args, "y");
    return max(x, y);
}
```

The first line checks to see if the name of the function passed to the *execute* method is "max". If it is, the parameters passed with the function (as *String* values in the *HashMap* "args") are converted to integer values and finally the *max* function is called – its return value is passed back from the *execute* method to the calling Worklet Service.

The *getArgAsInt* method called in the snippet above is defined in the *Implementation* section of the class's code. It is here that you can create private methods that carry out the external work of the any functions defined, as required.

The definition of *isNotCompleted* is slightly different, since the parameter passed is the special variable *this*. The *this* variable is essentially a *WorkItemRecord* that contains descriptors of the workitem the rule is testing, enabling developers to write methods that test the values in the variable and act on those values

accordingly. If it is for a case-level rule, *this* contains the case data for the instance invoking the rule. Both versions of *this* are passed as a string-ified JDOM Element format. See the YAWL source code for more details of the *WorkItemRecord* class, if required.

The *execute* method's sub-block for the *isNotCompleted* function looks like this:

```
if (name.equalsIgnoreCase("isNotCompleted")) {  
    String taskInfo = (String) args.get("this");  
    return isNotCompleted(taskInfo);  
}
```

The block gets the *this* variable as a *String* from the “args” *HashMap* and then calls the actual *isNotCompleted* method:

```
public static String isNotCompleted(String itemInfo) {  
    Element eItem = JDOMConversionTools.stringToElement(itemInfo);  
    String status = eItem.getChildText("status");  
    return String.valueOf(! isFinishedStatus(status) );  
}
```

Notice again that the function has been declared as *private static* and returns a *String* value. The first line of the function converts the *String* passed into the function to a JDOM Element, and then extracts from that Element a value for “status” (being one of the data attributes contained in the *this* variable). It then calls another method, defined in the *Implementation* section, called *isFinishedStatus*:

```
private static boolean isFinishedStatus(String status) {  
    return status.equals(WorkItemRecord.statusComplete) ||  
           status.equals(WorkItemRecord.statusForcedComplete) ||  
           status.equals(WorkItemRecord.statusFailed) ;  
}
```

All methods defined in the *Implementation* section must also be declared as *private static* methods – however, they can have any return type, so long as the value returned from the *execute* method back to the Worklet Service has been converted to a *String* value.

Of course, you are not restricted to querying the *this* variable as a *WorkItemRecord* – it is passed simply as a JDOM Element that has been converted to a *String* and so can be queried via a number of different methods.

The objective of the *RdrConditionFunctions* class is to allow developers to easily extend the capabilities of the Worklet Service by providing the means to test for other things in the conditional expressions of rule nodes other than the process instance's data attributes and values. It is envisaged that the class's functions can be extended into areas such as process mining, querying resource logs and external data sets.

Appendix B: Sample Log (generated by Walkthrough C)

```
2006-09-12 12:16:31,875 [INFO ] ExceptionService :- HANDLE CHECK CASE CONSTRAINT EVENT
2006-09-12 12:16:31,984 [INFO ] ExceptionService :- Checking constraints for start of case 20 (of specification: OrganiseConcert)
2006-09-12 12:16:32,093 [INFO ] ExceptionService :- No pre-case constraints defined for spec: OrganiseConcert
2006-09-12 12:16:32,109 [INFO ] ExceptionService :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 12:16:32,156 [INFO ] ExceptionService :- Checking pre-constraints for workitem: 20:BookStadium_5
2006-09-12 12:16:32,281 [INFO ] ExceptionService :- No pre-task constraints defined for task: BookStadium
2006-09-12 12:28:17,968 [INFO ] ExceptionService :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 12:28:18,000 [INFO ] ExceptionService :- Checking pre-constraints for workitem: 20:SellTickets_3
2006-09-12 12:28:18,015 [INFO ] ExceptionService :- No pre-task constraints defined for task: SellTickets
2006-09-12 12:28:18,078 [INFO ] ExceptionService :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 12:28:18,093 [INFO ] ExceptionService :- Checking post-constraints for workitem: 20.1:BookStadium_5
2006-09-12 12:28:18,093 [INFO ] ExceptionService :- No post-task constraints defined for task: BookStadium
2006-09-12 12:56:08,000 [INFO ] ExceptionService :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 12:56:08,015 [INFO ] ExceptionService :- Checking pre-constraints for workitem: 20:DoShow_4
2006-09-12 12:56:08,140 [INFO ] ExceptionService :- Workitem 20:DoShow_4 failed pre-task constraints
2006-09-12 12:56:08,140 [DEBUG] ExceptionService :- Invoking exception handling process for item: 20:DoShow_4
2006-09-12 12:56:08,156 [DEBUG] ExceptionService :- Exception process step 1. Action = suspend, Target = workitem
2006-09-12 12:56:08,171 [DEBUG] ExceptionService :- Successful work item suspend: 20:DoShow_4
2006-09-12 12:56:08,203 [DEBUG] ExceptionService :- Exception process step 2. Action = compensate, Target = ChangeToMidVenue
2006-09-12 12:56:08,343 [DEBUG] WorkletService :- Worklet specification 'ChangeToMidVenue' is already loaded in Engine
2006-09-12 12:56:08,546 [DEBUG] WorkletService :- Launched case for worklet ChangeToMidVenue with ID: 21
2006-09-12 12:56:08,578 [INFO ] ExceptionService :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 12:56:08,593 [INFO ] ExceptionService :- Checking post-constraints for workitem: 20.2:SellTickets_3
2006-09-12 12:56:08,593 [INFO ] ExceptionService :- No post-task constraints defined for task: SellTickets
2006-09-12 12:56:08,593 [INFO ] ExceptionService :- HANDLE CHECK CASE CONSTRAINT EVENT
2006-09-12 12:56:08,593 [INFO ] ExceptionService :- Checking constraints for start of case 21 (of specification: ChangeToMidVenue)
2006-09-12 12:56:08,609 [INFO ] ExceptionService :- No pre-case constraints defined for spec: ChangeToMidVenue
2006-09-12 12:56:08,609 [INFO ] ExceptionService :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 12:56:08,640 [INFO ] ExceptionService :- Checking pre-constraints for workitem: 21:CancelStadium_3
2006-09-12 12:56:08,656 [INFO ] ExceptionService :- No pre-task constraints defined for task: CancelStadium
2006-09-12 13:02:48,171 [INFO ] ExceptionService :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 13:02:48,187 [INFO ] ExceptionService :- Checking pre-constraints for workitem: 21:Book_Ent_Centre_5
2006-09-12 13:02:48,234 [INFO ] ExceptionService :- No pre-task constraints defined for task: Book_Ent_Centre
2006-09-12 13:02:48,250 [INFO ] ExceptionService :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 13:02:48,265 [INFO ] ExceptionService :- Checking post-constraints for workitem: 21.1:CancelStadium_3
2006-09-12 13:02:48,265 [INFO ] ExceptionService :- No post-task constraints defined for task: CancelStadium
2006-09-12 13:10:10,468 [INFO ] ExceptionService :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 13:10:10,484 [INFO ] ExceptionService :- Checking pre-constraints for workitem: 21:Tell_Punters_4
2006-09-12 13:10:10,500 [INFO ] ExceptionService :- No pre-task constraints defined for task: Tell_Punters
2006-09-12 13:10:10,500 [INFO ] ExceptionService :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 13:10:10,515 [INFO ] ExceptionService :- Checking post-constraints for workitem: 21.2:Book_Ent_Centre_5
2006-09-12 13:10:10,515 [INFO ] ExceptionService :- No post-task constraints defined for task: Book_Ent_Centre
2006-09-12 13:13:59,281 [INFO ] ExceptionService :- HANDLE CHECK CASE CONSTRAINT EVENT
2006-09-12 13:13:59,281 [DEBUG] ExceptionService :- Checking constraints for end of case 21
2006-09-12 13:13:59,281 [INFO ] ExceptionService :- No post-case constraints defined for spec: ChangeToMidVenue
2006-09-12 13:13:59,296 [DEBUG] ExceptionService :- Worklet ran as exception handler for case: 20
2006-09-12 13:13:59,437 [DEBUG] ExceptionService :- Exception process step 3. Action = continue, Target = workitem
```

```

2006-09-12 13:13:59,468 [DEBUG] ExceptionService :- Successful work item unsuspend: 20:DoShow_4
2006-09-12 13:13:59,515 [INFO ] ExceptionService :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 13:13:59,531 [INFO ] ExceptionService :- Checking post-constraints for workitem: 21.3:Tell_PunTERS_4
2006-09-12 13:13:59,531 [INFO ] ExceptionService :- No post-task constraints defined for task: Tell_PunTERS
2006-09-12 13:13:59,546 [INFO ] ExceptionService :- Exception monitoring complete for case 21
2006-09-12 13:13:59,750 [INFO ] ExceptionService :- HANDLE CHECK CASE CONSTRAINT EVENT
2006-09-12 13:13:59,875 [DEBUG] ExceptionService :- Checking constraints for end of case 20
2006-09-12 13:13:59,953 [INFO ] ExceptionService :- No post-case constraints defined for spec: OrganiseConcert
2006-09-12 13:14:00,046 [INFO ] ExceptionService :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 13:14:00,046 [INFO ] ExceptionService :- Checking post-constraints for workitem: 20.3:DoShow_4
2006-09-12 13:14:00,156 [INFO ] ExceptionService :- No post-task constraints defined for task: DoShow
2006-09-12 13:14:00,171 [INFO ] ExceptionService :- Exception monitoring complete for case 20

```